UNIVERSIDADE FEDERAL DE SANTA CATARINA

MOVIE.ME – UM SISTEMA DE RECOMENDAÇÃO DE FILMES BASEADO NO FACEBOOK

Mariah Barros Cardoso

Florianópolis – SC,
2016

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

MOVIE.ME – UM SISTEMA DE RECOMENDAÇÃO DE FILMES BASEADO NO FACEBOOK

Mariah Barros Cardoso

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

Florianópolis – SC,
2016

Mariah Barros Cardoso

MOVIE.ME – UM SISTEMA DE RECOMENDAÇÃO DE FILMES BASEADO NO FACEBOOK

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Orientadora:

_____

Profª. Drª. Carina Friedrich Dorneles

Banca examinadora:

_____

Prof. Dr. Leandro José Komosinski

_____

Prof. Dr. Roberto Willrich

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTAS DE ABREVIAÇÕES

AJAX – *Asynchronous Javascript and XML*

API – *Application Programming Interface*

HTML – *HyperText Markup Language*

HTTP – *Hypertext Transfer Protocol*

JSON – *JavaScript Object Notation*

MQL – *Metaweb Query Language*

RSS – *Really Simple Syndication*

XML – *eXtensible Markup Language*

RESUMO

Sistemas de recomendação tem sido amplamente utilizados por empresas e serviços visando oferecer ao usuário conteúdo que seja do seu interesse em meio ao enorme volume e variedade de informações disponíveis na Web. Apesar de bem-sucedidos ao longo do tempo, esses sistemas ainda enfrentam alguns desafios, sendo o *cold-start* um dos mais conhecidos. Como o nome sugere, esse problema é caracterizado pela falta inicial de dados que pode resultar em recomendações insatisfatórias. Diante desse cenário, este trabalho propõe o desenvolvimento de um sistema de recomendação de filmes que utiliza as informações disponíveis no perfil do usuário da rede social Facebook para reduzir o problema de *cold-start*. Objetivando o entendimento da solução proposta, são apresentados conceitos relacionados a sistemas de recomendação e redes sociais, além das tecnologias que foram utilizadas no desenvolvimento da aplicação. Ao final, o sistema tem seu desempenho avaliado por métricas de avaliação e teste envolvendo usuários reais.

Palavras-chave: sistema de recomendação, rede social, facebook, *cold-start*

1. INTRODUÇÃO

A rápida popularização e crescimento da Internet na última década fez dela a maior fonte de dados de acesso público existente no mundo (LIU, 2011). Devido a esse fato, o uso da Web como fonte de informação, entretenimento, cultura, produtos e serviços, tornou-se indispensável na rotina de grande parte da população. Diante desse contexto, os sistemas de recomendação surgiram com o objetivo de auxiliar o usuário a lidar com essa sobrecarga de informação, filtrando o conteúdo de forma personalizada de acordo com as preferências de cada usuário.

Desde o seu surgimento, os sistemas de recomendação têm sido utilizados por inúmeras empresas e serviços de diversos segmentos. No âmbito de entretenimento, são frequentemente empregados por serviços de *streaming* de áudio e vídeo. Um exemplo bastante conhecido é o Netflix[1], serviço de *streaming* de filmes e séries de TV que utiliza um algoritmo de recomendação bastante poderoso devido a enorme base de dados que possui. Estima-se que 70% dos vídeos assistidos por usuários ativos no sistema provenham das recomendações realizadas pelo algoritmo[2], o que demonstra a sua eficácia.

Apesar de bem-sucedidos, os sistemas de recomendação ainda enfrentam alguns desafios e limitações. Um dos mais conhecidos é o problema de *cold-start* que ocorre quando o sistema não possui dados suficientes sobre o usuário para realizar recomendações satisfatórias. No Netflix, por exemplo, isso acontece quando o usuário efetua o cadastro no serviço e ainda não assistiu e/ou avaliou uma quantidade mínima de vídeos, o que impossibilita o sistema de criar um perfil de interesse desse usuário e realizar as recomendações.

Diversas abordagens foram propostas pela literatura para resolver o *cold-start.* Entre as mais simplistas, está a que propõe realizar questionários prévios para obter dados explícitos do usuário. Essa abordagem é utilizada, por exemplo, pelo serviço de recomendação de filmes Movielens[3] que solicita ao usuário que, após efetuar o cadastro no sistema, classifique previamente um conjunto de filmes apresentados. Uma crítica feita a essa solução é que ela pode gerar desinteresse por parte do usuário devido ao acréscimo de etapas extras para uso do sistema. Uma outra solução proposta é a de Sahebi e Cohen (1997) que sugere a utilização

[1] https://www.netflix.com
[2] http://www.huffingtonpost.com/2013/08/01/netflix-profiles_n_3685876.html
[3] https://movielens.org/

de sistemas externos, em especial as redes sociais, para obter informações sobre o usuário, seus interesses, afinidades e interações sociais. Essa solução é a utilizada por este trabalho que tem como objetivo principal o desenvolvimento de um sistema de recomendação de filmes que utiliza os dados disponíveis no perfil do usuário da rede social Facebook[4].

O objetivo principal deste trabalho pode ser decomposto nos seguintes objetivos específicos:

- Obter as avaliações de filmes realizas por usuários do Movielens e coletar as informações necessárias referentes a esses filmes das base de dados IMDB[5] e TMDB[6];
- Implementar um sistema de recomendação utilizando as técnicas de recomendação colaborativa e baseada em conteúdo;
- Conectar o sistema a rede social Facebook, visando obter informações sobre o usuário alvo;
- Validar o sistema proposto através de experimentos *online* com usuários reais e experimentos *off-line* com métricas de avaliação;

Para atingir esses objetivos, é realizado um estudo referente a área de recomendação da informação através de conceitos recorrentes na literatura e obras relacionadas. Além disso, também são estudados outros conceitos relacionados as demais tecnologias que compõem o sistema, como as redes sociais.

Este trabalho se inicia no Capítulo 2, onde é tratada a fundamentação teórica utilizada como base para desenvolvimento do sistema proposto. No Capítulo 3, são apresentados três trabalhos relacionados ao tema deste trabalho e ao final do capítulo é realizada uma análise comparativa entre eles e o presente trabalho. No Capítulo 4, o sistema proposto e sua respectiva implementação são descritos mais detalhadamente. No Capítulo 5, é apresentada uma análise dos resultados obtidos no teste com usuários reais e com a avaliação *off-line*. Por fim, o Capítulo 6 apresenta as conclusões obtidas e trabalhos futuros.

[4] https://facebook.com/
[5] http://www.imdb.com/
[6] https://www.themoviedb.org/

2. FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são tratados os conceitos essenciais para a compreensão da ideia central do trabalho, assim como as técnicas que serviram como base para o desenvolvimento do sistema proposto.

2.1 Sistemas de recomendação

Os sistemas de recomendação tornaram-se uma importante área de pesquisa a partir da década de 90 (ADOMAVICIUS; TUZHILIN, 2005), impulsionados principalmente pela expansão do comércio eletrônico.

O primeiro sistema de recomendação, nomeado Tapestry, surgiu no início dos anos 90 e foi desenvolvido por pesquisadores da Xerox Palo Alto Research Center com o objetivo de lidar com o aumento no número de *e-mails* que chegavam no centro de pesquisa (GOLDBERD *et al*., 1992). Os idealizadores do Tapestry cunharam o termo “filtragem colaborativa” para designar o sistema, pois a filtragem da informação era realizada com auxílio de grupos de pessoas de mesmo interesse. Devido a isso, durante esse período, diversos pesquisadores acabaram por adotar essa terminologia para se referirem a qualquer tipo de sistema de recomendação. Posteriormente, Resnick e Varian (1997) propuseram um termo mais genérico, “sistema de recomendação”, argumentado ser o mais adequado pelo fato de existirem outras abordagens de recomendação que não são baseadas na colaboração entre usuários.

Atualmente, o interesse nesta área continua grande por ser rica em problemas de pesquisa e também pelo surgimento de diversos serviços que auxiliam o usuário a lidar com a sobrecarga de informação através da geração de conteúdo personalizado. (NUNES; CAZELLA, 2011).

2.1.1 Definição

Burke (2002) define sistema de recomendação como qualquer sistema que produza recomendações individualizadas ou que guie o usuário de forma personalizada para objetos de seu interesse dentre as diversas opções disponíveis. Também especifica que deve ser formado por três componentes: (1) dados prévios, ou seja, as informações que o sistema possui antes de iniciar as recomendações; (2)

dados de entrada, que consistem nas informações previamente fornecidas pelo usuário com o propósito de auxiliar no processo de recomendação; e (3) um algoritmo que processa os dados prévios e os dados de entrada a fim de gerar recomendações.

Uma definição formal de recomendação foi proposta por Adomavicius e Tuzhilin (2005) e consiste em: seja *C* um conjunto de todos os usuários do sistema, *S* um conjunto de todos os itens possíveis de serem recomendados e *u* uma função utilidade que mede o quão útil é um determinado item *s* para um usuário *c*, então: $u : C \times S \rightarrow R$, onde *R* é um conjunto totalmente ordenado. Para cada usuário $c \in C$, elege-se um item $s' \in S$ que maximiza a utilidade do usuário. Mais formalmente, isso pode ser expresso pela equação (1) proposta por Adomavicius e Tuzhilin (2005):

$$\forall c \in C, s' = \arg max_{s \in S}\, u\,(c, s) \qquad (1)$$

A utilidade de um item é normalmente expressa por um valor que indica o quanto determinado usuário aprova o item. Cada item *s* pode ser definido por um conjunto de características que variam dependendo do domínio da aplicação. No caso mais simples, o item pode conter apenas um identificador como atributo. Da mesma forma, cada usuário *c* também pode ter características associadas ou apenas um identificador.

O problema central dos sistemas de recomendação consiste no fato da função de utilidade *u* geralmente não ser definida para todo espaço *C x S* e sim apenas para um subconjunto desse. A utilidade costuma ser definida através de avaliações e essas são definidas apenas nos itens previamente avaliados pelos usuários. Dessa forma, o algoritmo de recomendação deve ser capaz de estimar as avaliações não realizadas para pares usuário-item e de fazer recomendações assertivas baseadas nessas predições.

Várias técnicas foram propostas na literatura para se estimar as avaliações desconhecidas como, por exemplo, métodos de aprendizagem de máquina, heurísticas e teorias de aproximação. Os sistemas de recomendação são geralmente classificados em função da técnica que utilizam para obterem essas estimativas. Uma vez obtidas, a maioria dos sistemas de recomendação selecionam aquelas com maiores valores para serem recomendadas para o usuário alvo.

2.1.2 Perfil do usuário

Cazella, Nunes e Reategui (2010) definem perfil do usuário como um conceito que reflete o interesse do usuário em relação a variados assuntos em determinado momento. Fisicamente é representado por uma base de dados que armazena informações sobre o usuário e suas preferências.

Dudev *et al.* (2008) sugere que a construção do Perfil do usuário deve seguir três etapas importantes:

- **Modelagem do usuário:** é avaliado e definido quais características do usuário são relevantes para o sistema e devem ser coletadas. Essas características podem ser dados demográficos, interesses e preferências, por exemplo.
- **Criação do perfil do usuário:** consiste na criação do perfil do usuário através da coleta e processamento das informações definidas na etapa anterior.
- **Personalização:** é a etapa final onde o perfil do usuário é utilizado pelo sistema para gerar recomendações personalizadas.

Durante a etapa de criação do perfil do usuário, as informações podem ser coletadas de forma explícita ou implícita. Na coleta explícita, as informações são obtidas de forma direta, ou seja, o usuário insere de forma explícita informações a seu respeito e suas preferências. Já na coleta implícita, o sistema infere as preferências do usuário através das suas ações e comportamento.

Uma das formas mais comuns de se obter dados explícitos é através da avaliação direta do usuário sobre determinado item. Essa avaliação pode ser realizada de forma escalar ou binária. A forma escalar, também conhecida como *ranking*, consiste de valores numéricos (1 a 5 estrelas, por exemplo) ou ordinais (concordo, neutro e discordo, por exemplo) que representam o nível de satisfação do usuário. Por outro lado, a forma binária apresenta apenas dois valores possíveis para o nível de satisfação (gosto ou não gosto, por exemplo).

No caso da coleta implícita, os dados podem ser obtidos de diversas formas, que variam dependendo do contexto que o sistema está inserido. Em sites de comércio eletrônico, por exemplo, são utilizados o histórico de compras do usuário, a quantidade de tempo gasto pelo usuário procurando determinado item, páginas de produtos que foram navegadas, entre outros.

A coleta explícita é considerada mais confiável e precisa, mas tem a desvantagem de depender da participação direta do usuário que nem sempre tem disponibilidade ou disposição para informar seus dados e preferências. Já a coleta implícita costuma ser mais fácil de ser obtida, apesar de ter um processamento mais difícil, pois cabe ao sistema definir se determinadas ações do usuário representam ou não suas preferências.

Na prática, as abordagens de coleta explícita e implícita também podem ser combinadas para se obterem melhores resultados.

2.1.3 Técnicas de recomendação

Os sistemas de recomendação têm sido implementados mediante o uso de diferentes técnicas que variam conforme o objetivo a ser atingido e ao tipo de item a ser recomendado.

O modelo de Burke (2002) apresenta cinco técnicas de recomendação: colaborativa, baseada em conteúdo, demográfica, baseada na utilidade e baseada no conhecimento. Dentre essas, esta seção aborda apenas as duas primeiras que foram as utilizadas na implementação do sistema deste trabalho, além de serem as mais utilizadas na prática. Além delas, também é apresentada a técnica de recomendação híbrida.

Recomendação colaborativa: a técnica de recomendação colaborativa, também chamada de filtragem colaborativa, surgiu a partir da observação de que, no geral, os indivíduos costumam confiar em recomendações fornecidas por outros indivíduos em tomadas de decisões rotineiras. Um exemplo disso, é quando você solicita a um amigo indicações de músicas ou filmes. A essência dessa técnica está justamente na troca de experiências entre os indivíduos que possuem interesses em comum (CAZELLA; NUNES; REATEGUI, 2010), ou seja, a ideia principal é de que usuários semelhantes costumam atribuir um *feedback* semelhante aos mesmos itens.

De acordo com Breese, Heckerman e Kadie (1998) os algoritmos de recomendação colaborativa podem ser categorizados em baseados em memória e baseados em modelo.

- **Algoritmos baseados em memória:** segundo Herlocker (1999), o funcionamento desse algoritmo pode ser dividido em três etapas: (1) cálculo do peso de cada usuário em relação ao usuário alvo utilizando uma métrica de similaridade; (2) seleção de um subconjunto de usuários com maiores níveis de similaridade, também chamados de “vizinhos” do usuário alvo, os quais serão utilizados para o cálculo da predição; (3) normalização das avaliações fornecidas pelos usuários para o item analisado na recomendação e cálculo da predição, ponderando-se as avaliações dos vizinhos com seus respectivos pesos de similaridade. O algoritmo mais utilizado nesta técnica é o do vizinho mais próximo (*K-Nearest Neighbor*). Os algoritmos baseados em memória são mais populares e simples. Uma desvantagem, porém, é que seu desempenho pode ser comprometido se a base de dados for muito grande, pois toda predição é processada utilizando toda base de dados.
- **Algoritmos baseados em modelo:** Em oposição aos baseados em memória, os algoritmos baseados em modelo aplicam algoritmos de mineração de dados para construir um modelo aproximado das avaliações atribuídas pelos usuários para predizer de forma inteligente novas avaliações. A construção desse modelo pode ser demorada, podendo levar até mesmo dias. O modelo resultante, porém, é pequeno, rápido e tão preciso como o baseado em memória. Segundo Breese, Heckerman e Kadie (1998) as técnicas mais utilizadas na construção dos modelos são as redes Bayesianas e o *clustering*. Os algoritmos baseados em modelo diminuem o tempo e o alto custo computacional durante a etapa de predição da recomendação quando comparados aos baseados em memória. Apesar disso, não são recomendados em ambientes onde as preferências dos usuários sofrem muitas mudanças, pois cada vez que isso ocorrer o modelo deverá ser reprocessado.

Uma das etapas mais importantes dos algoritmos baseados em memória é a de selecionar bons vizinhos. Tal importância se deve ao fato de afetar diretamente o tempo de execução, visto que essa é a etapa computacional mais cara desse tipo de algoritmo. Além disso, uma melhor seleção de vizinhos também afeta a acurácia, pois define os dados que serão utilizados na predição. Por essa razão, diferentes métricas são empregadas para calcular a similaridade entre os vizinhos, sendo a

eficácia de cada uma variável dependendo do algoritmo e tipo de dado utilizado. A seguir, são descritas as métricas mais conhecidas:

- **Distância Euclidiana:** na matemática representa a distância entre dois pontos. Para o contexto de sistemas de recomendação, essa ideia faz sentido ao se considerar os usuários como sendo pontos em um espaço de muitas dimensões, cuja coordenadas são os valores das suas avaliações. Ou seja, a distância euclidiana define a distância entre dois usuários. Dessa maneira, quanto menor o valor resultante, maior é a similaridade entre os usuários. Uma descrição formal apresentada por Manning, Raghavan e Schütze (2008) pode ser visualizada na equação (2), onde $w_{x,y}$ representa a similaridade entre o usuário *x* e o usuário *y*, x_i é a avaliação que o usuário *x* deu para o item *i* e y_i é a avaliação que o usuário *y* deu para o mesmo item *i*:

$$w_{x,y} = \sqrt{\sum_{i=1}^{M}(x_i - y_i)^2} \qquad (2)$$

- **Coeficiente de Correlação de Pearson:** é uma métrica que determina o grau de correlação entre duas variáveis quantitativas. Nos algoritmos baseados em memória, essas variáveis são representadas pelas linhas ou colunas da matriz de avaliações. O coeficiente varia entre os valores -1 e 1. O valor 0 expressa que não há relação linear, o valor 1 sugere uma relação linear perfeita e o valor -1 também indica uma relação linear perfeita, porém de forma inversa, ou seja, quando uma das variáveis diminui, a outra aumenta. Sendo assim, no âmbito de sistemas de recomendação, o valor 1 indica uma similaridade total entre os usuários e o -1 uma dissimilaridade total. A equação (3) apresenta uma definição formal do coeficiente de Pearson (CAZELLA; NUNES; REATEGUI, 2010):

$$corr_{ab} = \frac{\sum_i (r_{ai} - \overline{r_a})\,(r_{bi} - \overline{r_b})}{\sqrt{\sum_i i(r_{ai} - \overline{r_a})^2 \sum_i (r_{bi} - \overline{r_b})^2}} \qquad (3)$$

Onde, $corr_{ab}$ representa a correlação do usuário alvo *a* com outro usuário *b*; r_{ai} é a avaliação que o usuário *a* atribuiu para o item *i* e r_{bi} é a avaliação

que o usuário *b* atribuiu para o item *i*; r_a é a média de todas as avaliações do usuário *a* em comum com o usuário *b* e r_b é a média de todas as avaliações do usuário *b* em comum com o usuário *a.* Observa-se que para um resultado adequado, é necessário mais de uma avaliação em comum entre os usuários.

- ***Log-likelihood:*** essa métrica não utiliza as avaliações (notas) realizadas pelos usuários, apenas considera a associação dos itens com os usuários. A *log-likelihood* se baseia no número de itens em comum entre dois usuários para definir a similaridade entre eles. Primeiramente, é calculada a relação entre itens que estão associados a ambos usuários. Em seguida, é calculada a quantidade de itens que estão associados a um usuário, mas que não estão associados ao outro usuário. Por fim, é calculada a quantidade de itens que não estão associados a nenhum dos dois usuários. A matemática envolvida na definição formal dessa métrica está além do escopo deste trabalho.

Realizada a descrição das métricas de similaridade, é constatado que a distância euclidiana e o coeficiente de correlação de Person são mais adequados para contextos que possuem avaliações realizadas por usuários na forma quantitativa (formato de notas). Já a métrica *log-likelihood* é mais indicada para contextos onde as avaliações são booleanas, no qual apenas a associação usuário-item é considerada.

Recomendação baseada em conteúdo: na técnica de recomendação baseada em conteúdo, também conhecida como filtragem baseada em conteúdo, as recomendações são realizadas utilizando o conteúdo e descrição dos itens como referência. A ideia principal dessa técnica parte da observação de que, se um usuário tem preferência sobre determinado conteúdo hoje, ele tende a manter essa preferência durante algum tempo (HERLOCKER, 2000). No caso de um usuário, por exemplo, ter avaliado de forma positiva um filme do gênero comédia, a tendência natural é de que o sistema recomende outros filmes de comédia. Da mesma maneira, se o usuário avaliou um filme que possui em seu elenco diversos atores e alguns desses atores estão presentes também em outros filmes, o sistema pode inferir que o usuário possa vir a se interessar por outros filmes envolvendo esses atores.

O processo básico da recomendação baseada em conteúdo consiste, portanto, em cruzar os atributos do perfil do usuário com os atributos dos itens do sistema para, dessa forma, recomendar novos itens ao usuário. Uma das métricas mais utilizadas é a TF-IDF (*Term Frequency–Inverse Document Frequency*) que realiza comparação e cálculo de similaridade utilizando a frequência de ocorrência de palavras-chave no texto.

A abordagem baseada em conteúdo tem suas origens na área recuperação da informação, onde as primeiras aplicações implementadas faziam parte do domínio textual. Como consequência dos significativos avanços feitos pela comunidade de filtragem da informação e de conteúdo, muitos sistemas que utilizam a técnica baseada em conteúdo focam na recomendação de itens de conteúdo textual, como documentos e *sites* (CAZELLA; NUNES; REATEGUI, 2010).

Recomendação híbrida: consiste basicamente da combinação de duas ou mais técnicas de recomendação. O objetivo dessa abordagem é justamente o de aproveitar as vantagens de uma técnica para lidar com as limitações de outra (BURKE, 2002).

A grande maioria dos sistemas atuais de recomendação optam por utilizar mais de uma técnica de recomendação, sendo que a combinação mais utilizada é entre a filtragem colaborativa e a filtragem baseada em conteúdo, o que resulta em um melhor desempenho comparando-se ao uso exclusivo de uma só delas.

Segundo Adomavicius e Tuzhilin (2005) existem diferentes maneiras de combinar as duas técnicas, como é descrito a seguir:

- É implementado as duas técnicas de forma separada, sendo realizada apenas a combinação das recomendações que cada uma gerou;
- São incorporadas apenas algumas características da recomendação colaborativa em um sistema que utiliza recomendação baseada em conteúdo;
- São incorporadas apenas algumas características da recomendação baseada em conteúdo em um sistema que utiliza recomendação colaborativa. Essa foi a abordagem utilizada neste trabalho.
- É implementado um sistema unificado que incorpora simultaneamente as duas técnicas.

Apesar de oferecer vantagens, a recomendação híbrida requer um nível de complexidade mais elevado na implementação do sistema, pois exige que haja domínio de todas as técnicas envolvidas.

2.1.4 Desafios

Alguns dos principais desafios enfrentados pelos sistemas de recomendação são expostos por Adomavicius e Tuzhilin (2005) e Su e Khoshgoftaar (2009) e consistem em:

***Cold-start* (Partida a frio):** O problema de *cold-start* é dividido em duas categorias:

- ***Cold-start* de usuário:** também conhecido como problema do usuário novo, ocorre quando o sistema não possui dados iniciais suficientes sobre o usuário para que possa realizar recomendações satisfatórias. Tanto a técnica de filtragem colaborativa quanto a baseada em conteúdo sofrem com esse problema. Na filtragem colaborativa o *cold-start* ocorre quando o usuário, geralmente novo, avaliou poucos ou nenhum item do sistema. Dessa forma, o sistema não consegue encontrar outros usuários similares. Isso ocorre porque as métricas de similaridade, como o coeficiente de Person, necessitam que todos os usuários vizinhos tenham avaliado itens em comum para realizar a predição de um novo item. Já na filtragem baseada em conteúdo, o *cold-start* acontece quando o sistema não consegue criar um perfil de interesses do usuário, pois o mesmo não avaliou uma quantidade mínima de itens. A literatura indica algumas maneiras de solucionar o *cold-start*. Uma delas consiste em obrigar o usuário a realizar um número mínimo de avaliações para que possa usar o sistema, o que, apesar de eficaz, acrescenta etapas extras para o uso do sistema que pode resultar em uma percepção negativa do mesmo pelo usuário. Uma proposta melhor, é sugerida por Sahebi e Cohen (1997) que demonstra a eficácia do uso de dados extraídos de redes sociais para lidar com o *cold-start*.
- ***Cold-start* de item:** também chamado de problema do item novo, ocorre quando novos itens são adicionados ao sistema e acabam por não serem

recomendados até que algum usuário os avalie. Apenas a técnica de filtragem colaborativa sofre com o *cold-start* de item, visto que a filtragem baseada em conteúdo utiliza apenas as características do item para procurar por outros similares e não leva em consideração a avaliação de outros usuários. Uma maneira de solucionar o *cold-start* de item em um sistema que utiliza recomendação colaborativa é torna-lo híbrido e incorporar aspectos da recomendação baseada em conteúdo para tratar esse tipo de situação.

Superespecialização: esse problema ocorre em sistemas de recomendação baseado em conteúdo e é caracterizado pela incapacidade do sistema em recomendar itens que fujam do padrão de interesses do usuário. Desse modo, ao aprender os interesses do usuário e criar um perfil para o mesmo, o sistema não inova nas recomendações realizadas devido a pouca diversidade de itens que foram previamente avaliados ou interagidos pelo usuário.

Uma solução proposta por Adomavicius e Tuzhilin (2005) para reduzir a superespecialização é a de adicionar itens aleatórios entre os itens recomendados, apesar de não ser o ideal. Uma melhor maneira de contornar o problema seria tornar o sistema híbrido e incorporar aspectos de recomendação colaborativa para lidar com essa situação.

Análise de conteúdo limitada: na técnica de recomendação baseada em conteúdo é necessário que o conteúdo dos itens seja computado de alguma forma, seja automaticamente ou manualmente. No entanto, para alguns domínios esse processo pode se tornar mais dificultoso, como é o caso de informações em formato de imagem, vídeo e áudio. O mesmo não ocorre na recomendação colaborativa, visto que essa técnica não utiliza o conteúdo dos itens, apenas as suas avaliações.

Esparsividade: na maioria dos sistemas de recomendação colaborativos o número de avaliações efetuadas é geralmente muito menor do que o número de avaliações necessárias para se realizar a predição, ou seja, para esses sistemas é essencial a disponibilidade de uma massa crítica de usuários.

Um exemplo da consequência da esparsividade de dados é o caso de itens que foram avaliados por poucos usuários e por isso raramente são recomendados.

Isso contribui para que itens mais populares (com mais avaliações) tenham mais chances de serem recomendados que itens menos populares, o que nem sempre é o ideal.

Uma maneira de contornar esse problema é utilizando dados sobre os usuários no cálculo de similaridade. Dessa maneira, usuários podem ser considerados similares não apenas devido as suas avaliações, mas também devido as suas características em comum (dados demográficos, etc.).

***Grey Sheep* (Ovelha cinzenta):** esse problema afeta os sistemas de recomendação colaborativa e ocorre com usuários que possuem gostos atípicos ou muito diferentes dos outros usuários do sistema. Isso faz com que seja difícil encontrar usuários similares o que gera recomendações pouco eficazes. Também pode afetar sistemas de recomendação baseada em conteúdo, onde torna-se difícil de encontrar itens semelhantes aos avaliados pelo usuário alvo.

2.1.5 Métodos de Avaliação

Herlocker *et al*. (2004) propõem duas maneiras distintas para realizar a avaliação de sistemas de recomendação:

***Off-line*:** é realizada utilizando uma base de dados com as preferências dos usuários previamente adquiridas. Geralmente são base de dados de terceiros, como o Movielens. O funcionamento desse método de avaliação consiste em aplicar o algoritmo em estudo para prever as preferências dos itens e posteriormente analisar os resultados utilizando uma ou mais métricas de avaliação. As métricas comumente utilizadas são:

- **MAE (*Mean Absolute Error*):** a métrica MAE calcula o desvio absoluto médio entre o valor estimado pelo sistema de recomendação e o valor real obtido da avaliação realizada pelo usuário. Essa diferença resultante é encarada como o erro da predição (HERLOCKER *et al*., 2004). O valor MAE de cada predição deve ser minimizado visando uma recomendação mais adequada. A equação (4) apresenta uma definição formal do MAE, onde p_i constitui os valores estimados pelo sistema, r_i os valores reais das

avaliações dos usuários aos itens recomendados e *n* o número de itens considerados (CAZELLA *et al*., 2009):

$$MAE = \frac{\sum_{i=1}^{n} |p_i - r_i|}{n} \qquad (4)$$

- **RMSE (*Root Mean Square Error*):** essa métrica está relacionada ao MAE (HERLOCKER *et al*., 2004) e calcula a raiz quadrada do quadrado médio da diferença entre o valor estimado pelo sistema de recomendação e o valor real obtido da avaliação do usuário. Sua definição formal pode ser visualizada na equação (5), no qual p_i são os valores estimados pelo sistema de recomendação, r_i são os valores reais das avaliações dos usuários e *n* o número de itens considerados:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (p_i - r_i)^2}{n}} \qquad (5)$$

  Apesar de a RMSE ser semelhante a MAE, ela costuma ser mais utilizada por sistemas de recomendação no geral, pois penaliza erros maiores. Um exemplo disso é o Netflix Prize[7] que optou por utilizar o RMSE como métrica para avaliar as propostas dos competidores.

- **Precisão e Revocação (*Precision and Recall*):** essas métricas costumam ser utilizadas por sistemas de recuperação de informação. No contexto de sistemas de recomendação, a métrica de Precisão representa a relação entre o número de conteúdos que o usuário considera relevante e o número de conteúdos que foram recomendados. Ou seja, ela indica o quanto uma predição realizada é próxima da avaliação real feita pelo usuário. Já a Revocação é uma medida do número de conteúdos recomendados que indica a quantidade de itens que são de interesse do usuário que aparecem na lista de recomendações. Ambas métricas são definidas formalmente por Herlocker *et al*. (2004) e podem ser visualizadas nas equações (6) e (7):

[7] http://www.netflixprize.com/

$$Precision = \frac{N_{rs}}{N_s} \qquad (6)$$

$$Recall = \frac{N_{rs}}{N_r} \qquad (7)$$

A Precisão e Revocação são mais indicadas para sistemas de recomendação que possuem as preferências do usuário no formato booleano, enquanto as métricas MAE e RMSE são mais adequados para preferências expressas no formato de notas.

A maior parte do trabalho desenvolvido na avaliação de sistemas de recomendação foi focado na avaliação *off-line*. A vantagem desse método está na sua rapidez e economia ao possibilitar realizar avaliações em grande escala utilizando diferentes algoritmos e base de dados de forma simultânea.

Ainda assim, a avaliação *off-line* também apresenta duas importantes desvantagens. A primeira delas é a escassez natural de preferências nas bases de dados que limitam o número de itens que podem ser utilizados para realizar a avaliação. Isso ocorre porque não é possível avaliar o êxito de um item recomendado a um usuário se não existir uma preferência do mesmo usuário para esse item. A segunda desvantagem está no fato de que esse método de avaliação se limita a apenas predizer os resultados. Dessa forma, nenhuma análise *off-line* é capaz de avaliar a satisfação de usuários reais de um sistema, pois isso está intrinsecamente relacionado a fatores mais subjetivos como gosto e experiência com a interface, por exemplo.

Teste com usuários reais: a avaliação é realizada a partir da experimentação e observação do sistema sendo utilizado por usuários reais.

Esse método pode ser realizado de forma controlada ou não. Na forma controlada, são escolhidos os usuários que participarão da experimentação e em quais cenários eles serão submetidos. Já na forma não controlada pode ser realizado um estudo de campo onde o sistema é disponibilizado a um grupo de usuários que terão seus comportamentos observados.

Ao contrário da avaliação *off-line*, o teste com usuários reais sofre da falta de métricas padronizadas que possibilitam realizar comparações diretas entre

diferentes algoritmos. Esse fato, porém, é justificado pelo próprio objetivo desse tipo de método que está em analisar a interação do usuário com o sistema. Os resultados obtidos são, portanto, de ordem mais subjetiva e restritas ao domínio e contexto em estudo, entre outros fatores (CANHOTO, 2013).

Segundo Ricci *et al.* (2011) diferentes métodos de avaliações podem ser utilizados conforme o estágio de desenvolvimento do sistema. Durante a fase de projeto e implementação, o ideal é o método de avaliação *off-line* para auxiliar na escolha da técnica de recomendação e algoritmos. Após o sistema ter sido concluído, o método de teste com usuários reais é o mais adequado para validar o sistema.

2.2 Redes sociais

Recuero (2009, p. 24) descreve uma rede social como “um conjunto de dois elementos: atores (pessoas, instituições ou grupos; os nós da rede) e suas conexões (interações ou laços sociais) ”. Segundo Boyd e Ellison (2007) embora o propósito e nomenclatura utilizada variam de uma rede social para outra, todas apresentam uma estrutura básica que consiste em um perfil completamente ou parcialmente público do usuário onde consta seus dados pessoais e as suas conexões. O processo de inscrição geralmente é realizado mediante o preenchimento de dados em um formulário. A partir desses dados, o sistema gera um perfil associado ao usuário. O passo seguinte consiste em identificar os usuários já cadastrados no sistema e quais se deseja estabelecer uma conexão. Essas conexões podem ser tanto mútuas como unidirecionais (seguidores e fãs, por exemplo). As conexões mútuas geralmente exigem a aprovação de ambas as partes e se caracterizam por relações onde existe uma maior intimidade entre os usuários. De uma maneira geral, as redes sociais permitem que o usuário compartilhe conteúdo multimídia através do seu perfil e troque mensagens com outros usuários do sistema.

Nos últimos anos, as redes sociais são um fenômeno de grande sucesso entre os usuários da Web. Isso ocorre devido ao fato de haver uma interação entre as pessoas, onde elas podem expressar suas opiniões sobre determinado assunto ou situação. É esse o diferencial que as tornou tão populares e acessadas.

3. TRABALHOS RELACIONADOS

Neste capítulo, são apresentados trabalhos relacionados a área de recomendação da informação que utilizam dados de redes sociais para auxiliar no processo de recomendação. Ao final, é realizada uma análise comparativa entre esses trabalhos e o presente projeto.

3.1 O Que Tá Valendo? Um Sistema Web de Recomendação de Eventos

Procurar informações na Internet sobre os eventos que estão ocorrendo na cidade é uma prática cada vez mais comum. São inúmeros os sistemas Web e mobile desenvolvidos para a divulgação de eventos e até mesmo as redes sociais estão sendo utilizadas para esse propósito. A grande maioria desses sistemas, porém, apenas se limitam a agregar e informar os eventos agendados e não levam em consideração os gostos e preferências do usuário. Tal aspecto resulta em uma maior dificuldade por parte do usuário para identificar eventos que sejam de fato do seu interesse. Visando oferecer uma solução para esse problema, o trabalho de Busatto (2013) propõe o desenvolvimento do sistema Web “O que tá valendo?” que tem como propósito a recomendação personalizada de eventos culturais que estão acontecendo na cidade de Porto Alegre, utilizando para isso conceitos de recomendação e dados de grandes portais e do Facebook.

Para realizar a coleta de dados referente aos eventos, foi desenvolvido um módulo de *web crawling* que extrai as informações dos eventos em sites pré-definidos. Esse processo é realizado através da identificação no código fonte das *tags* HTML que possuem o conteúdo desejado. Como cada site selecionado possui um documento HTML único, foi necessário implementar um modelo de extração específico para cada um. Sendo assim, apesar do *web crawler* proposto ser bastante eficaz para a obtenção de dados automaticamente, ele é completamente dependente da estrutura do código de fonte, tornando-se necessária a correção das *tags* HTML de busca caso haja alguma alteração no mesmo.

O módulo de recomendação utiliza conceitos de recomendação colaborativa e baseada em conteúdo. Para sua implementação, foi necessário realizar a captura de dados explícitos e implícitos do usuário. Para obter os dados implícitos foi

desenvolvido o módulo de identificação e integração ao Facebook que é responsável por criar o cadastro de um novo usuário (ou realizar a identificação de um usuário prévio) no sistema, utilizando para isso o seu respectivo perfil no Facebook. Esse processo de autenticação é realizado através da API oficial do Facebook. Dessa forma, durante o primeiro *login* do usuário no sistema são obtidas as páginas curtidas por ele no Facebook. O sistema também adota outra alternativa para coletar dados implícitos que consiste em registrar o comportamento dos usuários, ou seja, toda vez que um usuário, seja ele cadastrado ou não, clica para obter mais informação de um evento é criado, através de uma requisição AJAX, um novo registro no banco de dados que contém o ID do usuário (ou 0, caso ele não seja cadastrado) e o ID do evento acessado. Para coletar os dados explícitos, o sistema permite que todo usuário cadastrado realize avaliações dos locais onde ocorrem os eventos conforme o seu interesse sobre os mesmos, atribuindo notas de 1 a 5 estrelas, representando de pouco a muito interesse, respectivamente.

A ideia central do módulo de recomendação é de que as pessoas costumam frequentar e se interessar por eventos que aconteçam em locais do seu agrado. Sendo assim, o sistema visa reconhecer quais são os locais de maior interesse do usuário cadastrado e identificar quais são os locais onde está havendo maior incidência de procura por eventos em um contexto geral (mais acessados, considerando todos os usuários do sistema) e, a partir de um processo de comparação e cálculo entre essas informações, recomendar os eventos que estarão acontecendo na cidade.

Através das informações obtidas e dos dados disponíveis no sistema, foram definidos nove critérios que são levados em consideração no cálculo da recomendação de eventos. Esses critérios podem ser visualizados no Quadro 1. O administrador do sistema pode definir pesos, de 1 a 10, para cada um dos critérios apresentados, podendo assim priorizar (ou não) no cálculo aqueles que forem do seu interesse.

Quadro 1 – Critérios definidos que são levados em consideração no cálculo da recomendação de eventos. UNC refere-se a um usuário não cadastrado e UC a um usuário cadastrado.

CRITÉRIO	UNC	UC
Eventos acessados por qualquer usuário, com qualquer data de acontecimento e data de acesso ao sistema.	Sim	Sim
Eventos acessados por qualquer usuário que ocorrerão posteriormente à	Sim	Sim

data de acesso ao sistema.		
Eventos acessados pelo usuário cadastrado com qualquer data de acontecimento e data de acesso ao sistema.	Não	Sim
Eventos acessados pelo usuário cadastrado que ocorrerão posteriormente à data de acesso ao sistema.	Não	Sim
Eventos acessados por qualquer usuário que ocorrerão na data de acesso ao sistema.	Sim	Sim
Eventos acessados pelo usuário cadastrado que ocorrerão na data de acesso ao sistema.	Não	Sim
Avaliação dos locais realizada por qualquer usuário.	Sim	Sim
Avaliação dos locais realizada pelo usuário cadastrado.	Não	Sim
Análise das páginas curtidas pelo usuário cadastrado no Facebook	Não	Sim

Fonte: Adaptado de Busatto (2013, p. 41-45)

O cálculo utilizado para realizar a recomendação de eventos pode ser dividido em quatro passos: o cálculo dos critérios relacionados a acessos, o cálculo dos critérios relacionados a avaliação de locais, o cálculo dos critérios da análise das páginas curtidas no Facebook e a soma dos valores obtidos.

No primeiro passo são considerados os seis primeiros critérios do Quadro 1. Para cada um desses seis critérios, é obtido primeiramente o total de acessos separados por local, através dos eventos acessados. Depois, é obtido o total de eventos acessados. Com esses totais é realizado o cálculo do percentual de acessos que cada local obteve. Esse percentual é então multiplicado pelo peso definido para o critério. O valor resultando é armazenado em um vetor bidimensional indexado pelo ID do local e o ID do critério. Vale ressaltar que cada critério define se serão considerados acessos de todos os usuários do sistema ou apenas do usuário cadastrado que o está utilizando e se serão considerados eventos que ocorrem em qualquer data ou apenas na data de acesso ao sistema.

O segundo passo utiliza os critérios sete e oito do Quadro 1. Para o critério sete, é obtida a média das avaliações do local. Já para o critério oito, é obtido o valor da avaliação realizada pelo usuário cadastrado para o dado local. Para os dois critérios, o valor obtido é multiplicado por 20 e pelo peso do critério. O valor resultante da multiplicação é então armazenado em um vetor bidimensional indexado pelo ID do local e ID do critério.

O terceiro passo utiliza o último critério do Quadro 1. Nesse passo é obtido da base de dados os registros das páginas que o usuário cadastrado curtiu no

Facebook. Para cada local cadastrado no sistema é verificado se o usuário curtiu a sua página no Facebook. Caso verdadeiro, retorna o valor 100. Caso contrário, o valor 0. Esse valor é multiplicado pelo peso do critério. O valor resultante é então armazenado em um vetor bidimensional indexado pelo ID do local e o ID do critério.

Ao fim desses três passos, é obtido um vetor indexado pelo ID do local e os nove valores resultantes do cálculo de cada critério. O próximo e último passo é a soma dos valores de cada local e o reordenamento do vetor de forma decrescente. Os índices das primeiras posições do vetor representam os ID dos locais onde provavelmente haverá mais interesse do usuário sobre os eventos que lá ocorrerão. Esses IDs são utilizados em uma consulta ao banco de dados que retorna os eventos que ocorrerão na data selecionado pelo usuário, ordenados por seus locais de preferência. O sistema também adota a técnica de *Treemapping* para a visualização da informação dos eventos recomendados, onde é concedido maior espaço na tela para itens que possuam maior importância no contexto.

Apesar de não ter sido realizado a avaliação de uso por usuários reais, o autor considera que o sistema proposto atingiu o seu objetivo, apesar de algumas limitações, principalmente no módulo de *crawling* que, quando executado, consome todos os recursos do servidor no processamento e acaba impedindo o uso do sistema por outros usuários durante esse período.

3.2 SuggestMe - Um Sistema de Recomendação Utilizando Web Semântica para Evitar o *Cold Start*

Um dos principais e mais conhecidos desafios enfrentados pelos sistemas de recomendação é o *cold-start*. O trabalho de Formiga (2014) apresenta um protótipo de recomendação de bandas e artistas denominado SuggestMe que soluciona o problema de *cold-start* obtendo dados do Facebook e realizando recomendações utilizando web semântica através do Freebase.

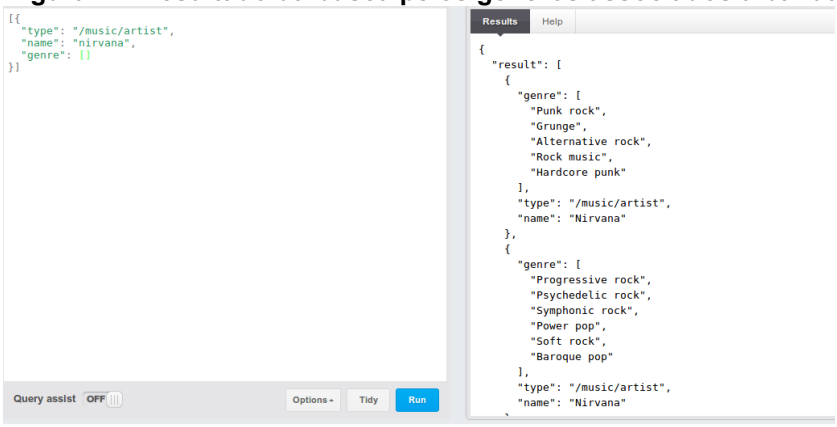
O Freebase[8] é uma base de dados colaborativa que possui dados das mais variadas fontes e dispõe de uma grande quantidade de informações. Especificamente para o protótipo proposto foram utilizadas apenas as informações de bandas e artistas. Para se ter acesso aos dados do Freebase é necessário

[8] https://www.freebase.com/

utilizar a linguagem MQL que retorna *strings* em JSON contendo o resultado da consulta.

O funcionamento básico do sistema pode ser descrito da seguinte forma: primeiramente o usuário deve realizar o *login* no sistema utilizando a sua conta no Facebook. Dessa maneira, é possível obter através da API oficial do Facebook todas as informações públicas do perfil do usuário, incluindo as páginas curtidas de bandas e artistas. Tendo essas informações, o sistema realiza consultas no Freebase utilizando a MQL para descobrir os gêneros associados a cada banda/artista curtida pelo usuário. Um exemplo dessa busca pode ser visualizado na Figura 1.

Figura 1 – Resultado da busca pelos gêneros associados a banda Nirvana.



Fonte: Formiga (2014, p. 14).

Em seguida, o sistema cria uma matriz de comparação que guarda gênero e valor. Caso o gênero ainda não exista na matriz, ele é adicionado com o valor da *flag* 1. Caso contrário, é apenas incrementado o valor 1 no campo *flag*. Um exemplo de matriz pode ser visualizado no Quadro 2.

Quadro 2 – Matriz de comparação de gêneros musicais.

GÊNERO	FLAG
Punk Rock	1
Alternative Rock	2
Funk	1
Hard Rock	4
Reggae	3
Trance	4

Fonte: Adaptado de Formiga (2014, p. 28)

Após finalizar o preenchimento da matriz de comparação, é realizada uma ordenação decrescente considerando o campo *flag* visando obter os gêneros que tiveram mais ocorrências, ou seja, os que o usuário tem mais afinidade. Em seguida, o sistema seleciona os *n* gêneros que têm o maior valor na coluna *flag*, sendo o valor *n* igual ao número de bandas/artistas curtidas pelo usuário Exemplificando: se o usuário só curtiu 4 bandas e essas bandas foram associadas a 30 gêneros na matriz, o sistema irá selecionar apenas os 4 gêneros com maior número na coluna *flag* da matriz de comparação. Os resultados obtidos dos gêneros de maior afinidade do usuário são inseridos então em um vetor de gêneros. O próximo passo é buscar outras bandas com base nesse vetor de gêneros. Novamente, são realizadas buscas no Freebase utilizando a MQL passando como parâmetro individualmente cada gênero do vetor. O resultado retornado é o de bandas/artistas que possuem o gênero em questão associado. Como o conjunto de resultados é muito grande, é selecionado apenas 1 resultado de cada gênero. Todos os resultados dessas consultas são salvos em outro vetor chamado vetor resultado que é utilizado para mostrar os resultados finais ao usuário. Por fim, é sugerido vídeos mais assistidos de cada banda/artista presente no vetor resultado utilizando para esse fim a API do Youtube[9]. Os dados das recomendações feitas para o usuário são salvos no banco de dados do sistema para utilização futura.

Ao final do trabalho são apresentados os resultados obtidos de usuários que utilizaram o sistema. No geral, o sistema obteve sucesso em suas recomendações, porém, enfrentou algumas dificuldades. Uma delas ocorre quando o usuário do sistema curtiu muitas bandas brasileiras no Facebook, pois o Freebase possui poucas informações de bandas brasileiras. O funcionamento do sistema também depende diretamente das quantidades de páginas de bandas/artistas que o usuário curtiu no Facebook, quanto maior esse número, melhores serão as sugestões. Se o usuário não tiver curtido nenhuma banda/artista o sistema não será capaz de realizar recomendações. Como trabalho futuro, é sugerido utilizar um cálculo de similaridade mais elaborado, que use dados de outros usuários (amigos na rede social), além da utilização de uma outra base de dados conjuntamente que possua mais informações de bandas nacionais.

[9] https://www.youtube.com/

3.3 Usando o Twitter para Recomendar Notícias em Tempo-Real

O surgimento da Web facilitou de forma significativa o acesso a informação por parte do usuário, em especial no que se refere as notícias divulgadas pelos mais diversos meios de comunicação. Diante desse cenário, sistemas de recomendação de notícias ganharam popularidade devido a sua capacidade de filtrar conteúdo relevante para o usuário em meio ao excesso de informação. O artigo de Phelan, McCarthy e Smyth (2009) descreve um protótipo chamado Buzzer que representa uma nova abordagem na recomendação de notícias e utiliza a rede social Twitter como referência para selecionar conteúdo de interesse do usuário contido em uma coleção de *feeds* RSS.

Feed RSS e Twitter são duas importantes tecnologias da Web 2.0. O primeiro é uma tecnologia que foi projetada para fornecer acesso a conteúdos atualizados com frequência. Nele é informado um pequeno resumo da notícia e seu respectivo link. Um leitor de RSS permite ao usuário agregar vários *feeds* RSS e tem como função fornecer notícias atualizadas de diversas fontes em uma só interface. Em contraste, o Twitter é uma rede social e um servidor para *microblogging* onde o usuário pode realizar postagens curtas chamadas *tweets* (máximo de 140 caracteres) e seguir postagens de outros usuários. Nos últimos tempos, tem havido um grande interesse no Twitter, não só devido ao seu sucesso, mas também por esse fornecer acesso a informações em tempo real originárias de milhões de usuários.

O principal objetivo do Buzzer é justamente o de aproveitar o diferencial do Twitter em relação aos outros meios de comunicação o qual consiste em oferecer conteúdo sobre os acontecimentos quase que no mesmo instante em que eles ocorrem, além do seu caráter mais informal que contribui para uma maior diversidade de temas que possivelmente possam estar mais relacionados ao contexto de vida do usuário.

O Buzzer adota uma técnica de recomendação baseada em conteúdo que tem sido bem-sucedida ao longo do tempo, conforme é demonstrado por Pazzani e Billsus, 2007. O sistema compreende três componentes básicos:

- ***Configuration Interface*:** responsável por gerenciar o processo de cadastro e *login* do usuário onde o mesmo fornece informações sobre a sua conta no Twitter e informa *feeds* RSS que deseja seguir. É importante

ressaltar que se o usuário não tiver uma conta no Twitter, o sistema irá utilizar a linha do tempo pública do Twitter.

- ***Lucene Indexer*:** é responsável pela mineração e indexação das informações oriundas do Twitter e do RSS segundo as configurações do usuário.
- ***Recommendation Engine*:** gera uma lista ordenada de artigos RSS com base na co-ocorrência de termos populares contidos no conteúdo do Twitter e RSS. O processo pela qual é gerada a lista ordenada é apresentado da seguinte forma:

Dado um usuário *u* e um conjunto de *feeds* RSS *r*, o sistema primeiramente extrai os artigos RSS mais recentes *R* e *tweets* do Twitter *T*. Separadamente indexa cada artigo e *tweet* para produzir dois índices Lucene. Os termos do índice resultante são então extraídos desses índices RSS e Twitter como base para produzir os vetores de termos RSS e Twitter (*MR* e *MT*, respectivamente).

Em seguida, é identificado o conjunto de termos *t* que co-ocorrem em *MT* e *MR* que são as palavras que estão presentes nos últimos *tweets* e nos mais recentes artigos RSS e são usadas como base para a técnica de recomendação. Cada termo t_i é usado em uma consulta realizada no índice RSS para recuperar o conjunto de artigos *A* que contém t_i juntamente com sua pontuação TF-IDF associada. Desse modo, cada co-ocorrência de t_i é associada ao conjunto de artigos *A1,…An* o qual contém t_i e a pontuação TF-IDF para t_i em cada *A1,…An* para produzir uma matriz.

Para calcular uma pontuação global para cada artigo, é apenas calculado a soma das pontuações TF-IDF através de todos os termos associados com esse artigo, como é demonstrado pela equação (8) abaixo:

$$Score(A_i) = \sum_{\forall t_i} element(A_i, t_i) \qquad (8)$$

Dessa maneira, os artigos que contêm muitos *tweet* termos com alta pontuação TF-IDF são preferíveis aos artigos que contêm *tweet* termos com baixa pontuação TF-IDF. É um mecanismo de pontuação simples, porém suficiente para fornecer um ponto de partida.

A recomendação resultante é simplesmente realizada através da seleção dos top K artigos com as maiores pontuações.

O Buzzer também permite que o usuário escolha via interface gráfica entre três estratégias distintas de recomendação que são:

- ***Public-Rank*:** é utilizada a linha do tempo pública do Twitter ao invés da conta pessoal do usuário.
- ***Friends-Rank:*** são mineradas as postagens de amigos do Twitter.
- ***Content-Rank*:** nessa estratégia, o Twitter é desconsiderado e é apenas realizado a ordenação dos artigos baseando-se na frequência dos termos RSS sozinhos.

O sistema também gera uma nuvem de *tags* baseada na frequência de cada termo.

Para avaliar as recomendações geradas pelo Buzzer, foi conduzida uma avaliação preliminar com um pequeno grupo de 10 pessoas durante o período de 5 dias. Cada participante configurou o sistema informando até 10 dos seus *feeds* RSS favoritos e sua conta no Twitter. Periodicamente, eles selecionavam diferentes estratégias de recomendação. Como medida de validação foi optado por focar na frequência de cliques que os artigos recebiam conforme a estratégia de recomendação utilizada. Os resultados obtidos demonstram uma clara diferença no comportamento dos usuários quando é optada por uma das duas estratégias que utilizam o Twitter em comparação com a que não utiliza. Em média, as estratégias baseadas no Twitter resultaram entre 8,3 e 10,4 cliques por usuário em comparação com apenas 5,8 cliques na estratégia que desconsidera o Twitter. Isso resultou em um aumento de 30% a 45% no número de cliques. A análise de cliques também demonstrou uma preferência da estratégia *Friends-Rank*, o que sugere que os usuários são mais propensos a entrar em sintonia com temas de interesse dos seus amigos do que do público em geral. Curiosamente, no entanto, 67% dos participantes informaram preferir a estratégia *Public-Rank* e apenas 22% indicaram preferir a *Friends-Rank*, o que vai de encontro ao resultado da análise realizada através do número de cliques. Vale ressaltar, porém, que nenhum participante disse preferir a estratégia de *Content-Rank,* enquanto 11% disseram não saber que estratégia preferiam. Apesar das duas análises gerarem resultados distintos, ambas confirmam que os usuários são beneficiados quando se utiliza o Twitter para realizar

as recomendações, demonstrando assim a significância do estudo realizado neste artigo.

3.4 Quadro comparativo

O Quadro 3 apresenta uma análise comparativa entre os sistemas dos trabalhos relacionados abordados anteriormente e do sistema proposto por este trabalho.

Quadro 3 – Comparação entre os sistemas dos trabalhos relacionados e o Movie.Me.

CARACTERÍSTICAS	O QUE TÁ VALENDO?	SUGGESTME	BUZZER	MOVIE.ME
Tipo de conteúdo a ser recomendado	Eventos	Bandas e artistas	Notícias	Filmes
Redes sociais utilizadas	Facebook	Facebook	Twitter	Facebook
Fonte de dados	Sites de eventos e Facebook	Freebase e Facebook	Feeds RSS	Movielens, IMDB, TMDB e Facebook
Forma que foram coletados os dados	*Web crawler* e Facebook Graph API	API do Freebase e Facebook Graph API	API do Twitter	MyAPIFilms e Facebook Graph API
Técnica de recomendação utilizada	Filtragem colaborativa e baseada em conteúdo	Filtragem baseada em conteúdo	Filtragem baseada em conteúdo	Filtragem colaborativa e baseada em conteúdo
Algoritmo Utilizado	Do autor	Do autor	TF-IDF	Do Mahout
Avaliações realizadas do sistema	Não foram realizadas avaliações	Testes com usuários reais	Testes com usuários reais	Avaliação off-line utilizando métricas de avaliação e testes com usuários reais

Através do Quadro 3, é possível observar que o sistema Buzzer é o único que não utiliza a rede social Facebook como fonte de informações.

Apesar dos sistemas analisados possuírem várias características em comum, o Movie.Me se difere dos demais pelo tipo de conteúdo recomendado, fonte de

dados empregadas, algoritmo utilizado para gerar as recomendações e pela validação do sistema utilizando métricas de avaliação. Entre todos apresentados, o sistema “O que tá valendo?” é o que mais se assemelha a solução proposta neste trabalho, pois ambos utilizam a abordagem de filtragem híbrida (colaborativa e baseada em conteúdo) e o Facebook como fonte de dados. A diferença fundamental, porém, está no algoritmo utilizado por cada um e na forma como o cálculo da recomendação é efetuado, o que acaba por gerar resultados bastante distintos. Além disso, ao contrário do Movie.Me, o sistema de Busatto (2013) não foi validado através de métricas de avaliação ou teste com usuários reais.

4. MOVIE.ME

Neste capítulo, é descrito o sistema proposto neste trabalho. Inicialmente, é exposta uma visão geral do sistema, fonte de dados utilizadas, sua arquitetura e módulos. Ao final, é apresentada a implementação do sistema e tecnologias utilizadas, além de um detalhamento de cada módulo.

4.1 Visão geral

O Movie.Me é um sistema *web* de recomendação de filmes que utiliza os dados do perfil do Facebook do usuário alvo como alternativa para reduzir o problema do *cold-start*. É baseado essencialmente na técnica de filtragem colaborativa, mas também incorpora elementos da filtragem baseada em conteúdo visando incrementar a qualidade de suas recomendações. Tal abordagem, como vista no Capítulo 2, é descrita por Adomavicius e Tuzhilin (2005) como uma técnica de recomendação híbrida. As fontes de dados empregadas, sua arquitetura e módulos são expostos ao longo desta seção.

4.1.1 Fonte de dados

As fontes de dados utilizadas pelo Movie.Me são detalhadas a seguir.

Movielens: é um sistema de recomendação colaborativo de filmes desenvolvido em 1997 pelo grupo de pesquisa Grouplens liderado por John Riedll da Universidade de Minnesota dos EUA. Para utilizar o sistema, os usuários devem realizar um cadastro e avaliar no mínimo quinze filmes, atribuindo-lhes uma nota de escala de 1 a 5. Essas avaliações iniciais são necessárias devido ao problema de *cold-start*. Feito isso, o sistema apresenta uma lista dos cinco melhores filmes recomendados para o usuário.

O Movielens disponibiliza sua base de dados com as avaliações dos usuários para ser utilizada por outros sistemas de recomendação para fins de teste e pesquisa. São disponibilizadas base de dados de diversos períodos e tamanhos. A utilizada pelo Movie.Me é a do período de janeiro de 2016 e composta por 105.339 avalições de 10.329 filmes feitas por 668 usuários.

IMDB: o Internet Movie Database é um portal *web* e base de dados *online* sobre filmes, séries de televisão e celebridades criado em 1990. Atualmente pertence a empresa Amazon[10] e é um dos mais completos e populares *sites* do gênero.

Foi utilizado pelo Movie.Me como fonte principal para se obter os seguintes dados dos filmes: títulos em diferentes idiomas, gêneros, países de origem, ano, duração, sinopse em inglês, número de votos, nota, filmes similares, atores e diretores.

TMDB: o The Movie Database surgiu em 2008 com a proposta de ser uma comunidade de compartilhamento de *posters* e *fanart* de filmes. A partir de 2009 alteraram seu foco e lançaram sua primeira base de dados de filmes. Atualmente, possui uma das maiores bases de dados sobre filmes, séries de televisão e celebridades, além de contar com uma comunidade de usuários bastante ativa.

Foi utilizado pelo Movie.Me como fonte secundária para se obter as sinopses em português e *posters* dos filmes.

Facebook: é, hoje, a maior e mais popular rede social da Web, tendo alcançado a incrível marca de um bilhão de usuários ativos em 2012[11]. Desde a sua fundação em 2004 por Mark Zuckerberg e seus colegas da faculdade de Havard, uma média de 316 mil pessoas se cadastram por dia na rede social[12]. No Brasil, ele é usado por oito em cada dez dos mais de 107 milhões de internautas do país[13].

A rede social exige que seus usuários possuam cadastro no site para que possam acessar e utilizar o sistema. Após realizar o cadastro, é criado um perfil onde o usuário pode informar dados pessoais, adicionar outros usuários como amigos e interagir com eles de diversas maneiras, seja através de mensagens, fotos, vídeos ou jogos. Um recurso também presente no sistema que se tornou bastante popular é o botão “curtir” pelo qual os usuários podem demonstrar gostar de certos conteúdos, tais como publicações de amigos, comentários, fotos, entre

[10] https://www.amazon.com/
[11] http://economia.estadao.com.br/noticias/negocios,facebook-ultrapassa-marca-de-1-bilhao-de-usuarios,129481e
[12] http://www.ultracurioso.com.br/5-opcoes-secretas-do-facebook-que-voce-nao-conhecia/
[13] http://g1.globo.com/tecnologia/noticia/2014/08/oito-cada-dez-internautas-do-brasil-estao-no-facebook-diz-rede-social.html

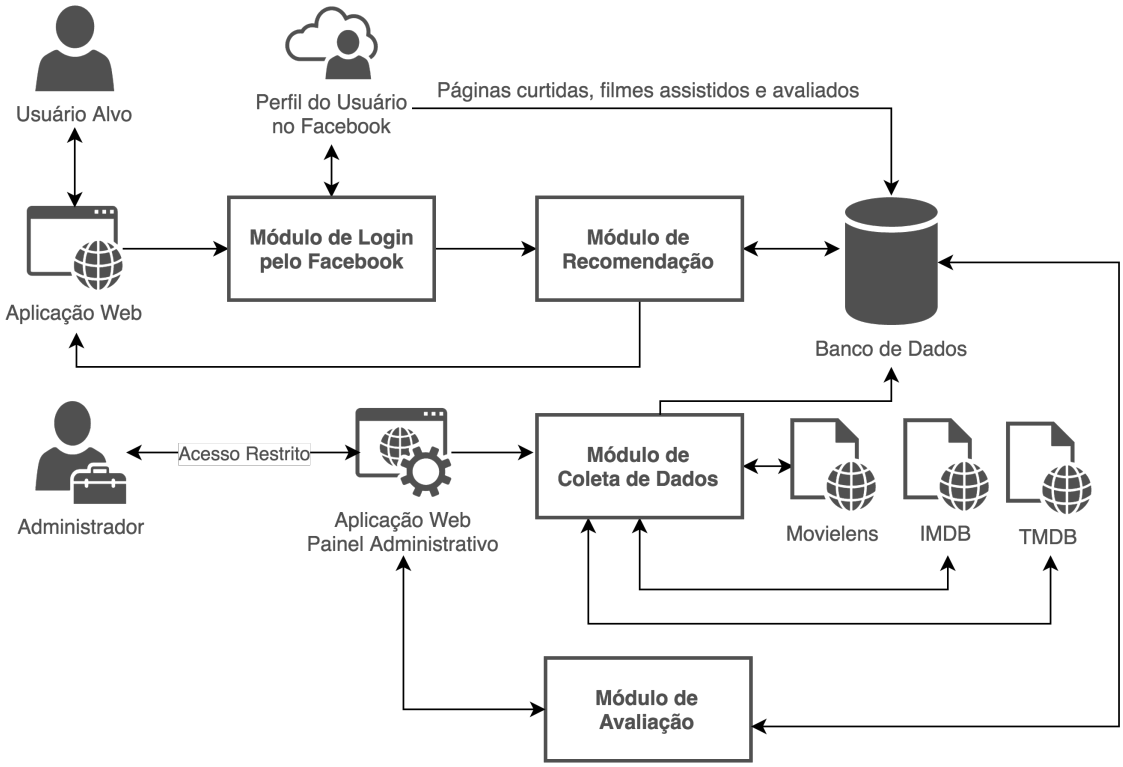
outros. Além disso, o usuário também pode curtir uma “página” do Facebook[14] que consiste em um perfil público criado por empresas, organizações, artistas, entre outros, com o propósito de divulgar informações e de criar um vínculo com os usuários interessados.

O Movie.Me utiliza as páginas curtidas pelo usuário e os filmes marcados como assistidos e avaliados presentes no seu perfil do Facebook.

4.1.2 Arquitetura

A arquitetura do sistema Movie.Me pode ser visualizada na Figura 2.

Figura 2 – Arquitetura do sistema.



Conforme apresentando na Figura 2, o Movie.Me é composto por quatro módulos e é acessado por uma interface *web*. A descrição de cada módulo é realizada na seção seguinte. Na Figura 2 também é possível observar que nem todos os módulos podem ser acessados pelos usuários alvo do sistema, sendo dois deles restritos apenas ao usuário administrator.

[14] https://www.facebook.com/about/pages

4.1.3 Módulos

Nesta seção, são descritos os módulos que compõem a arquitetura do sistema apresentada na Figura 2.

Módulo de coleta de dados: é encarregado de popular o banco de dados do Movie.Me com as avaliações de filmes realizadas pelos usuários do sistema Movielens e para cada filme avaliado, coletar e armazenar no banco de dados suas respectivas características (sinopse, duração, etc.) advindas das bases de dados IMDB e TMDB.

O acesso a este módulo pode ser realizado apenas pelo usuário administrador do Movie.Me e sua execução deve ser efetuada antes dos demais módulos, uma vez que os dados resultantes desse processo são necessários para o funcionamento dos outros módulos que compõem o sistema.

Módulo de *login* pelo Facebook: é responsável por *logar* o usuário no Movie.Me utilizando para tal a sua respectiva conta no Facebook. Sendo assim, torna-se um pré-requisito para o uso do sistema, o usuário possuir um cadastro prévio no Facebook.

Se for a primeira vez que o usuário realiza o *login* no Movie.Me, será solicitado a sua permissão para que sejam acessados os dados disponíveis no seu perfil do Facebook. Após ter sido concedida a permissão, o módulo de *login* irá extrair as informações do perfil do usuário que são consideradas relevantes para o propósito do sistema e salvá-las no banco de dados. É importante ressaltar que se o usuário não autorizar o acesso aos dados, ele ficará impossibilitado de solicitar por recomendações de filmes.

Caso o usuário já tenha realizado o *login* anteriormente, ou seja, utilizado o sistema em algum momento anterior, o módulo de *login* irá apenas autenticá-lo, não sendo realizada uma atualização dos dados do usuário. Isso ocorre em razão do fato de o sistema proposto neste trabalho tratar-se apenas de um protótipo, tendo sido julgado desnecessário implementar esse tipo de funcionalidade em tais circunstâncias.

Uma outra tarefa que este módulo também desempenha é a de efetuar uma limpeza nos dados oriundos do Facebook, eliminando conteúdo considerado irrelevante e padronizando a formatação dos dados.

A Figura 3 apresenta a tela do Movie.Me que solicita o *login* do usuário pelo Facebook.

Figura 3 – Tela do sistema que solicita o *login* pelo Facebook.

Módulo de Recomendação: tem como objetivo utilizar as informações previamente coletadas pelos módulos descritos anteriormente para gerar recomendações personalizadas de filmes ao usuário do Movie.Me.

A ideia principal do módulo se baseia na técnica de recomendação colaborativa e consiste em buscar por usuários do Movielens que tenham avaliações em comum com o usuário do Movie.Me a fim de utilizá-los como referência para realizar as recomendações. Além disso, também são empregadas características dos filmes como gêneros, país de origem, etc. no cálculo de recomendação, incorporando, dessa maneira, o conceito de recomendação baseada em conteúdo. O módulo também analisa as páginas curtidas no Facebook que possam ser relacionadas de forma indireta a determinados filmes na base de dados, como é o caso das páginas que representam atores, diretores e artistas em geral. Se o usuário do sistema curtiu a página de um determinado ator, por exemplo, isso também será considerado no cálculo da recomendação, que aumentará o peso dos filmes que contém o ator em questão.

Este módulo só pode ser acessado por usuários *logados* no sistema, sendo, portanto, a execução do módulo de *login* pelo Facebook um pré-requisito. Após o

módulo de recomendação ser executado, são exibidos ao usuário dez filmes recomendados, listados em ordem decrescente por relevância, conforme pode ser visualizado na Figura 4. Para cada filme exibido, é possível visualizar detalhes sobre o mesmo clicando em seu respectivo título ou pôster, o que contribui para um julgamento mais assertivo do usuário a respeito da recomendação.

Figura 4 – Tela do sistema com parte dos filmes recomendados para o usuário.

Módulo de Avaliação: tem como tarefas executar a avaliação *off-line* do Movie.Me e possibilitar o envio de *feedbacks* por parte do usuário, o que é fundamental para o teste com usuários reais. Para a avaliação *off-line*, o módulo disponibiliza diversas métricas de avaliação que são utilizadas para realizar testes no sistema com o propósito de ajustar os parâmetros do módulo de recomendação de forma que produza melhores resultados. Já para o teste com usuários reais, são disponibilizados botões para aprovar/reprovar um filme recomendado e um formulário onde o usuário pode assinalar o que achou das recomendações no geral (excelente/bom/regular/ruim) e, opcionalmente, enviar um comentário.

O acesso aos resultados da avaliação *off-line* é restrito, sendo permitido apenas para o usuário administrador do sistema. Já os botões de aprovação/reprovação e o formulário são acessíveis apenas para usuários *logados* no sistema e que tenham solicitado por recomendações de filmes.

4.2 Implementação

Nesta seção são apresentadas as tecnologias utilizadas pelo sistema, seus respectivos diagramas de classes e de entidade e relacionamento, bem como o processo de desenvolvimento de cada módulo do mesmo.

4.2.1 Tecnologias utilizadas

O sistema foi desenvolvida utilizando a linguagem de programação orientada a objetos Java[15], criada na década de 90 pela empresa Sun Microsystems. O motivo da escolha se deve ao fato de ser uma linguagem popular, poderosa e por possuir uma grande quantidade de bibliotecas disponíveis para uso público.

Para a base de dados, foi escolhido o MySQL [16] da empresa Oracle Corporation devido a sua popularidade e facilidade de integração.

As demais tecnologias utilizadas para o desenvolvimento do sistema proposto são detalhadas a seguir.

Play[17]**:** é um *framework* livre e de código aberto focado no desenvolvimento de aplicações web de forma ágil e eficiente. Foi criado em 2007 por Guillaume Bort, mas teve sua primeira versão publicada apenas em 2009. Sua versão 1.x foi desenvolvida em Java e a partir da versão 1.1 também passou a fornecer suporte a linguagem funcional Scala. Em 2011, Bort se juntou a empresa Typesafe e lançou a versão 2.x que foi totalmente reescrita em Scala. Ainda assim, o Play continua a oferecer suporte para o Java, apesar de existir uma preferência da comunidade pelo uso da linguagem Scala.

O Play apresenta como arquitetura o padrão *Model, View, Controller* (MVC) e ao contrário de *frameworks* populares como o Spring e o Struts, não é um *framework* padrão Java EE. Uma aplicação JSF, por exemplo, roda sobre a API de Servlet, que por sua vez roda em um container Java EE, que fica dentro de um HTTP Server, o que resulta em quatro camadas. Já o Play possui apenas duas camadas: o próprio Play e o servidor HTTP Netty embutido (BOAGLIO, 2014).

No sistema web proposto neste trabalho foi utilizado o Play *framework* visando facilitar o processo de desenvolvimento do mesmo.

[15] https://www.java.com/pt_BR/
[16] https://www.mysql.com/
[17] https://www.playframework.com/

Apache Mahout[18]**:** é uma biblioteca de aprendizagem de máquina de código aberto escrito em Java focado em recomendação, *clustering* e classificação. Como o nome sugere, foi criado pela Apache Software Foundation em 2008 como parte do projeto Apache Lucene[19], outra ferramenta também de código aberto focada em busca e recuperação da informação. Mais adiante, o Mahout absorveu o projeto Taste, um projeto de código aberto focado em filtragem colaborativa e em 2010 tornou-se um projeto independente. Foi desenvolvido visando também a sua integração com o Apache Hadoop[20], *software* voltado a computação distribuída. Ainda que a integração com o Hadoop seja recomendada e necessária ao lidar com uma expressiva quantidade dados, ela não é obrigatória, podendo o Mahout ser utilizado de forma não distribuída.

A API de recomendação do Mahout é voltada para a técnica de filtragem colaborativa e não oferece implementações diretas da técnica de filtragem baseada em conteúdo. Apesar disso, existem maneiras indiretas de implementar a recomendação baseada em conteúdo através da utilização de pontos de extensão da API do Mahout.

Neste trabalho, foi optado por utilizar o Mahout no desenvolvimento do módulo de recomendação. Como a computação distribuída foge do escopo deste trabalho e a quantidade de dados envolvida é razoável, a integração com o Hadoop foi julgada desnecessária e não foi realizada.

Facebook Graph API: o Facebook disponibiliza para desenvolvedores o acesso a sua API oficial chamada Graph API[21]. A partir dela, é possível realizar, através de requisições HTTP, consultas e ações no Facebook, como recuperar dados do usuário e as páginas curtidas por ele, publicar em sua linha do tempo, entre diversas outras tarefas. Para executar todas essas tarefas, porém, é necessário ter a permissão do usuário do Facebook. A Graph API utiliza o OAuth como protocolo de autorização.

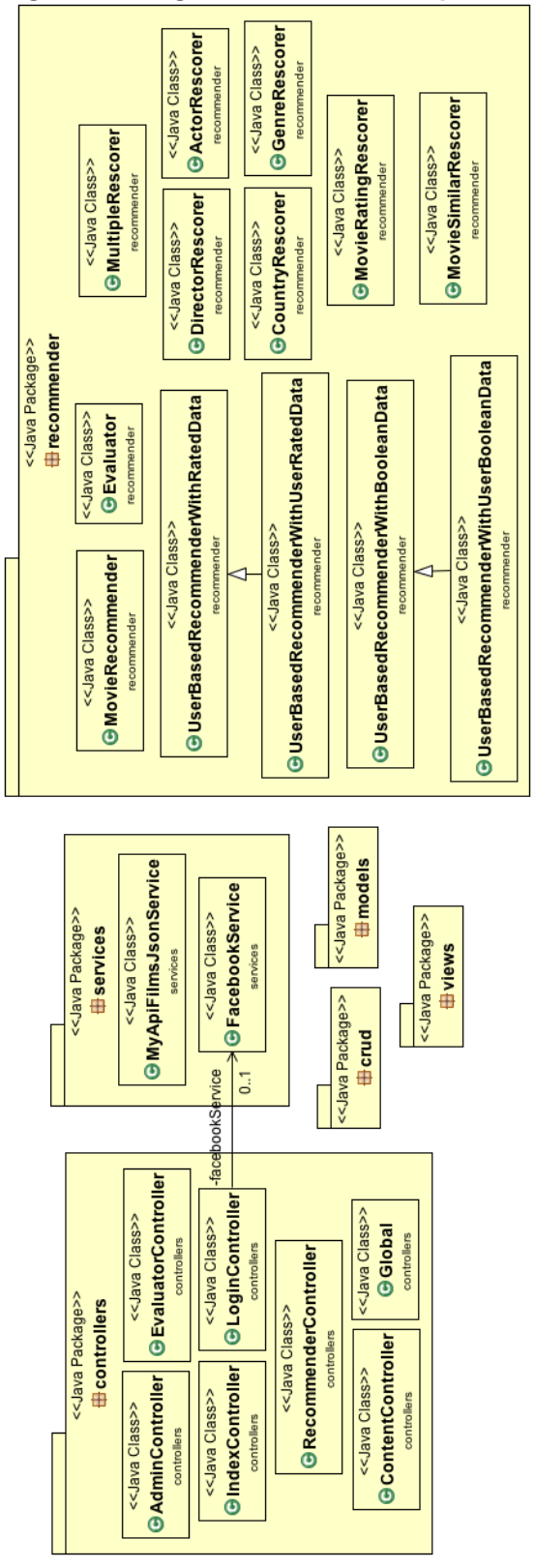
A Graph API é utilizada neste trabalho para realizar o *login* no sistema através do Facebook e também para coletar as informações pessoais e páginas curtidas do usuário.

[18] http://mahout.apache.org/
[19] https://lucene.apache.org
[20] http://hadoop.apache.org/
[21] https://developers.facebook.com/docs/graph-api

4.2.2 Diagrama de classes

A Figura 5 apresenta o diagrama de classes simplificado do sistema Movie.Me.

Figura 5 – Diagrama de classes simplificado do sistema.



As classes dos pacotes *crud*, *models* e *views* foram omitidas na Figura 5 por motivo de falta de espaço na imagem devido a grande quantidade de classes que esses pacotes englobam. A seguir, é realizada uma descrição de cada pacote presente no diagrama apresentado:

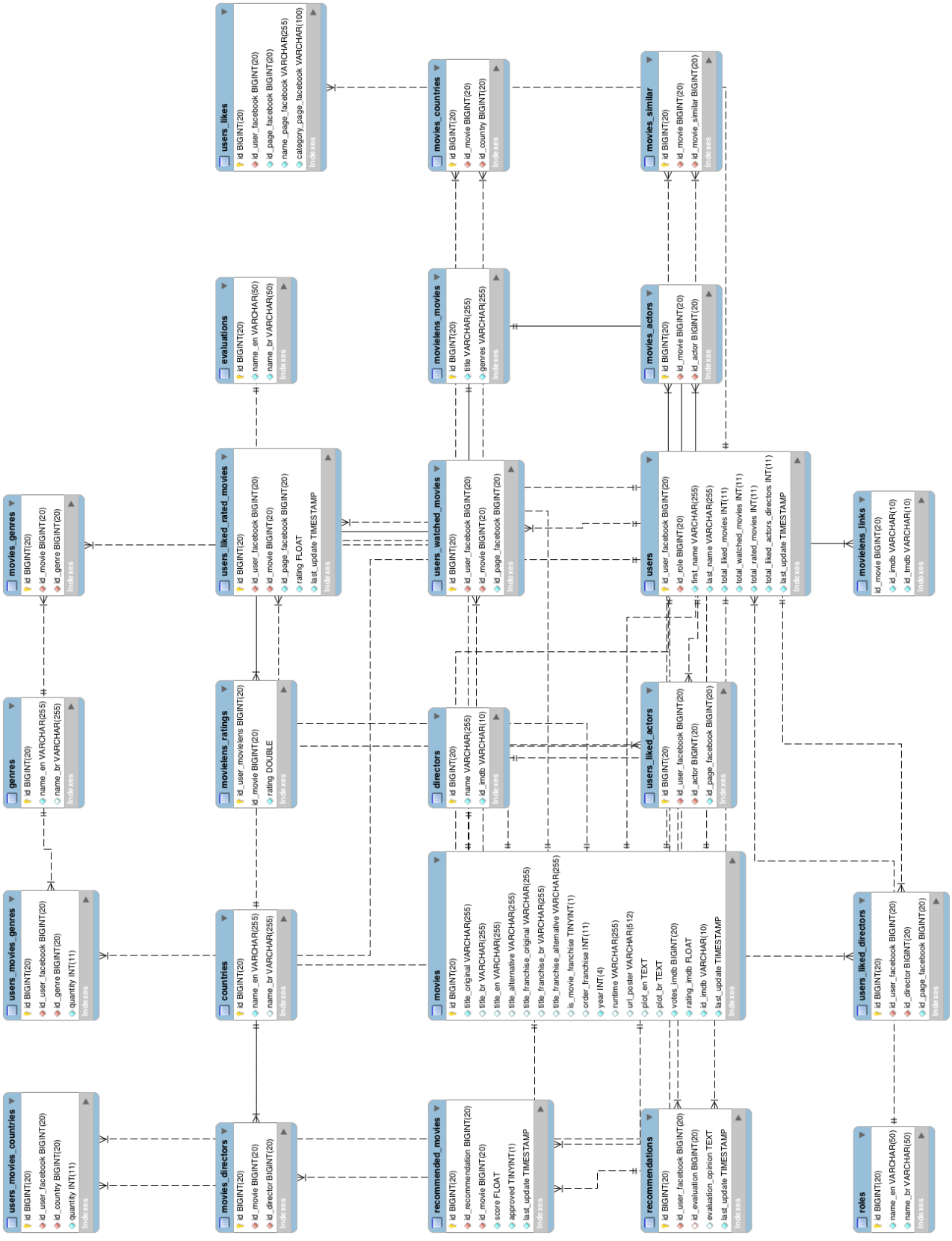
- ***controllers:*** corresponde as classes da camada de controle do Play *framework*, conforme o padrão MVC.
- ***models:*** inclui as classes responsáveis pelo mapeamento do banco de dados e corresponde a camada de modelo do padrão MVC.
- ***views:*** corresponde as classes da camada de visão do padrão MVC.
- ***crud:*** compreende as classes responsáveis pela manipulação do banco de dados.
- ***recommender:*** abrange as classes encarregadas de efetuar o processo de recomendação.
- ***services:*** contém as classes necessárias para extrair e manipular os dados do Facebook, IMDB e TMDB.

As classes utilizadas por cada módulo do sistema, bem como suas funções, são descritas em detalhes no decorrer da seção 4.2, com exceção das classes dos pacotes *models* e *crud*. Apesar disso, é importante salientar que essas classes foram utilizadas pelos módulos em todos os processos que envolvem manipulação do banco de dados.

4.2.3 Esquema do banco de dados

O esquema do banco de dados do Movie.Me pode ser visualizado na Figura 6 que corresponde ao modelo de entidade e relacionamento.

Figura 6 – Modelo de entidade e relacionamento do sistema.



O modo como cada tabela do banco de dados foi utilizada pelo Movie.Me e a forma como os dados presentes nas mesmas foram obtidos é descrito ao longo da seção 4.2, durante o detalhamento da implementação de cada módulo.

4.2.4 Módulo de coleta de dados

O Movielens concede acesso a sua base de dados através de arquivos no formato CSV que podem ser baixados no site do Grouplens[22]. São disponibilizados quatro arquivos:

- ***ratings.csv*:** composto pelas avaliações de filmes realizadas por usuários do Movielens. As notas atribuídas a cada filme possuem um intervalo de 0 a 5;
- ***movies.csv*:** contém o título dos filmes avaliados em inglês e seus respectivos gêneros;
- ***links.csv*:** relaciona o id de cada filme avaliado aos seus respectivos ids na base de dados do IMDB e TMDB;
- ***tags.csv*:** possui *tags* que os usuários do Movielens atribuíram a cada filme avaliado.

Neste trabalho, só foram utilizados os três primeiros arquivos, sendo o arquivo de *tags* descartado. Para cada um desses três arquivos, foi criada uma tabela correspondente no banco de dados e realizada a importação dos dados.

A etapa seguinte consiste em coletar informações referentes a cada filme importado da base de dados do Movielens utilizando para essa finalidade as bases de dados do IMDB e TMDB. Devido ao fato do IMDB não disponibilizar nenhuma API oficial para baixar os seus dados, apenas arquivos de texto simples, optou-se por utilizar a API não-oficial MyAPIFilms[23] para esse propósito. Ao contrário do IMDB, o TMDB possui uma API oficial, contudo a mesma não foi utilizada, visto que a MyAPIFilms também engloba o TMDB.

O funcionamento da MyAPIFilms pode ser descrito da seguinte forma:

- Primeiramente, deve-se realizar um cadastro no seu respectivo site para se obter um *token* de acesso.

[22] http://grouplens.org/datasets/movielens/
[23] http://www.myapifilms.com/

- Adquirido o *token*, basta realizar uma requisição a URL especificada pela API, utilizando o id do IMDB do filme que se deseja obter informações ou o seu título em determinado idioma. Adicionalmente, deve-se complementar a URL com parâmetros que identificam as informações que devem ser retornadas. Vale ressaltar que existe uma URL própria para consultas no IMDB e outra para consultas no TMDB. Em ambos os casos, o id do IMDB do filme pode ser utilizado para realizar a consulta. O título do filme, por outro lado, só pode ser utilizado na consulta do IMDB.
- Concluída a requisição, a MyAPIFilms irá retornar as informações do filme em questão no formato JSON (ou em XML, caso for solicitado).

O módulo de coletas de dados do Movie.Me realiza essas requisições de forma automática. As classes envolvidas nesse processo são:

- ***MyAPIFilmsJsonService.java* (do pacote *services*):** é onde são especificadas as URLs para realizar as consultas e no qual são efetuadas as requisições e manipulação do JSON retornado.
- ***AdminController.java* (do pacote *controllers*):** contém métodos responsáveis por realizar a coleta das informações dos filmes utilizando a classe *MyAPIFilmsJsonService* e por salvá-las na base de dados.
- ***admin.scala.html* (do pacote *views*):** contém a interface gráfica que permite ao usuário administrador do sistema solicitar a importação das informações dos filmes para a base de dados, caso a mesma ainda não tenha sido efetuada.

Ao final desse processo, as informações coletadas foram salvas nas seguintes tabelas do banco de dados do Movie.Me:

- ***actors***: contém o nome e id do IMDB dos principais atores presentes nos filmes.
- ***countries***: inclui o nome em português e em inglês dos países de origem dos filmes.
- ***directors***: possui o nome e id do IMDB dos diretores dos filmes.
- ***genres***: contém o nome em português e em inglês dos gêneros dos filmes.
- ***movies***: engloba todos os filmes da base de dados e suas respectivas características: títulos original, em português, em inglês e alternativo, ano,

duração, URL com a imagem do pôster, sinopse em inglês, sinopse em português, número de votos no IMDB, nota no IMDB e id do IMDB. Os atributos referentes a franquia correspondem aos filmes que fazem parte de uma série de filmes (como 007 e O Senhor dos Anéis, por exemplo). O título da franquia e a ordem de lançamento de cada filme da franquia foi inserido na base de dados de forma manual para as franquias de filmes mais famosas e populares. O motivo de não ter sido automatizado tal processo, é pelo fato de esse tipo de informação não ser disponibilizada nas bases de dados utilizadas como fonte de dados (IMDB e TMDB) e também por ser algo complexo e fora do escopo deste trabalho programar o reconhecimento inteligente dos filmes que fazem parte de uma franquia ou não.

- ***movies_actors:*** relaciona o id dos filmes com o id dos atores.
- ***movies_countries:*** relaciona o id dos filmes com o id dos países de origem.
- ***movies_directors:*** relaciona o id dos filmes com o id dos diretores.
- ***movies_genres:*** relaciona o id dos filmes com o id dos gêneros.
- ***movies_similar:*** relaciona o id dos filmes com o id dos filmes que são considerados similares de acordo com o IMDB.

4.2.5 Módulo de *login* no Facebook

Para auxiliar no desenvolvimento dos recursos deste módulo, foi utilizado o *plugin* para Play *framework* nomeado Play-Pac4j e a API em Java RestFB. Ambos se baseiam na Graph API para se comunicar com o Facebook.

O Play-Pac4j é encarregado de autenticar o usuário no Movie.Me através da sua conta no Facebook e adquirir um *token* de acesso que será necessário nas operações com a Graph API. Esse token é gerado a partir das permissões que o usuário autorizou ao realizar o *login*. Ou seja, caso o usuário não autorize nenhuma permissão, esse token só possuirá permissão para executar operações da Graph API que envolvam apenas o perfil público. No caso do Movie.Me, são solicitadas duas permissões: (1) curtidas: referente a todas as páginas curtidas pelo usuário no Facebook; (2) atividade de vídeo: engloba todos os vídeos marcados como assistidos e/ou que foram avaliados pelo usuário no seu perfil do Facebook.

A RestFB também é utilizada durante o processo de *login*, entretanto, apenas para extrair os dados do Facebook do usuário. Essa extração é realizada através de métodos da API RestFB que utilizam o *token* de acesso adquirido pelo Play-Pac4j.

As classes que compõem o módulo de *login* no Facebook são listadas a seguir:

- ***IndexController.java* (do pacote *controller*):** responsável por detectar se o usuário está *logado* ou não no sistema e restringir acesso as páginas dependentes de *login*. Caso o usuário não esteja *logado*, efetua o *login* utilizando métodos do Play-Pac4j. Se for um usuário novo, a classe *LoginController* é chamada para realizar o cadastro do mesmo no sistema.
- ***FacebookService.java* (do pacote *services*):** contém métodos que são encarregados de extrair as páginas curtidas e filmes assistidos e/ou avaliados pelo usuário no Facebook utilizando a RestFB para esse propósito. Também realiza uma limpeza nos dados extraídos do Facebook, que incluí os títulos das páginas e atributos das mesmas. Tal limpeza é essencial, visto que os títulos das páginas serão utilizados pela classe *LoginController* para encontrar os filmes/atores/diretores correspondentes no banco de dados do Movie.Me. Dessa forma, são excluídas palavras que são frequentemente colocadas nos títulos das páginas e que não fazem parte do nome do filme/ator/diretor, como por exemplo “O Filme”, “3D”, “Oficial”, “Fãs”, etc. Além disso, muitas páginas de filmes do Facebook não contêm os atributos de diretor e data de lançamento preenchidos. Devido a isso, também é realizada uma busca por possíveis datas presentes no título da página. Ainda que o atributo de data de lançamento seja preenchido, o seu formato costuma ser aleatório, o que também é tratado por essa classe que extraí o ano de lançamento de diferentes formatos de data. De maneira geral, toda essa limpeza é executada utilizando expressões regulares.
- ***LoginController.java* (do pacote *controller*):** é composta por métodos que efetuam o cadastro do usuário no sistema. Durante esse processo, são salvos no banco de dados os seguintes dados (se existirem) advindos do Facebook do usuário: (1) filmes curtidos, assistidos e/ou avaliados; (2) diretores e atores curtidos; (3) gêneros e países de origem mais recorrentes dos filmes coletados. Para localizar no banco de dados os

filmes correspondentes a página de um filme do Facebook é realizada primeiramente uma busca pelo título original do filme. Caso não encontre, a busca é feita pelo título em português e assim em diante para os títulos em inglês e alternativo. Se a página do filme no Facebook possuir os atributos de data de lançamento e diretor preenchidos (ou se foi extraído o ano de lançamento do título da página), a busca é executada incluindo esses atributos. Também existe um caso especial para os filmes que fazem parte de franquias, como por exemplo Star Wars, Harry Potter, Star Trek, entre outros. Nesse caso, é realizada uma busca também pelo título da franquia correspondente ao título da página do Facebook. Se encontrado, é inferido pelo sistema que a página se refere a uma franquia e, dessa forma, todos os filmes da franquia são associados ao usuário em questão. Vale ressaltar, porém, que no caso de o título da página corresponder a apenas um dos filmes da franquia (como Toy Story 3, por exemplo), isso não irá ocorrer, pois a página do Facebook se refere a apenas um filme da franquia, sendo assim, apenas o filme em questão associado ao usuário.

- ***login.scala.html* (do pacote *views*):** contém a interface gráfica para o usuário realizar o *login* no sistema Movie.Me.
- ***index.scala.html* (do pacote *views*):** página inicial do sistema Movie.Me que pode ser visualizada pelos usuários *logados* e onde é possível solicitar por recomendações de filmes.

Para que este módulo funcionasse também foi necessário criar um aplicativo no Facebook utilizando a sua interface para desenvolvedores[24], conforme pode ser visualizado na Figura 7. Tal etapa é exigida pelo Facebook para que se possa realizar o *login* utilizando o mesmo. O aplicativo do Facebook é conectado a este módulo através da classe *Global.java* (do pacote *controllers*), onde são informados para o *plugin* Play-Pac4j o id do aplicativo e seu código secreto. Complementarmente, é também configurado nessa classe quais permissões serão solicitadas ao usuário pelo Play-Pac4j durante o processo de *login*.

[24] https://developers.facebook.com/

Figura 7 – Interface do Facebook para desenvolvedores criarem aplicativos.

Após o cadastro do usuário ter sido efetuado com sucesso, seus dados são salvos nas seguintes tabelas do banco de dados do Movie.Me:

- ***users:*** contém o id do Facebook de cada usuário, seu nome e sobrenome e número total de filmes curtidos/assistidos/avaliados e atores e diretores curtidos.
- ***users_liked_actors:*** relaciona o id do Facebook de cada usuário com o id do Movie.Me dos atores curtidos por ele.
- ***users_likes:*** inclui todas as páginas curtidas por cada usuário, incluindo o id no Facebook de cada página, seu nome e categoria.
- ***users_liked_directors:*** relaciona o id do Facebook de cada usuário com o id do Movie.Me dos diretores curtidos por ele.
- ***users_liked_rated_movies:*** relaciona o id do Facebook de cada usuário com o id do Movie.Me dos filmes curtidos por ele. Também inclui o id no Facebook da página de cada filme e a nota que o usuário atribuiu a ele (de forma explícita – filme avaliado no Facebook com uma nota de 1 a 5, ou de forma implícita – se o filme foi curtido no Facebook pelo usuário, o Movie.Me atribui uma nota 5).
- ***users_movies_countries:*** relaciona o id do Facebook de cada usuário com o id do Movie.Me dos países de origem de todos os filmes associados ao usuário. Também atribui uma quantidade de incidência de cada país por usuário, para o sistema ter conhecimento de quais países de origem são mais recorrentes para cada usuário.

- ***users_movies_genres:*** relaciona o id do Facebook de cada usuário com o id do Movie.Me dos gêneros de todos os filmes associados ao usuário. Também atribui uma quantidade de incidência de cada gênero por usuário, para o sistema ter conhecimento de quais gêneros são mais recorrentes para cada usuário.
- ***users_watched_movies:*** relaciona o id do Facebook de cada usuário com o id do Movie.Me dos filmes assistidos por ele. Também inclui o id no Facebook da página de cada filme.

4.2.6 Módulo de recomendação

Para implementar este módulo foi escolhida a biblioteca Apache Mahout, devido as facilidades que oferece e pelo seu bom desempenho.

Conforme já mencionado anteriormente, a API de recomendação do Mahout é focada na técnica de filtragem colaborativa, entretanto permite incorporar aspectos da técnica de filtragem baseada em conteúdo no seu cálculo de recomendação. Essa abordagem parcialmente híbrida é a adota por este módulo.

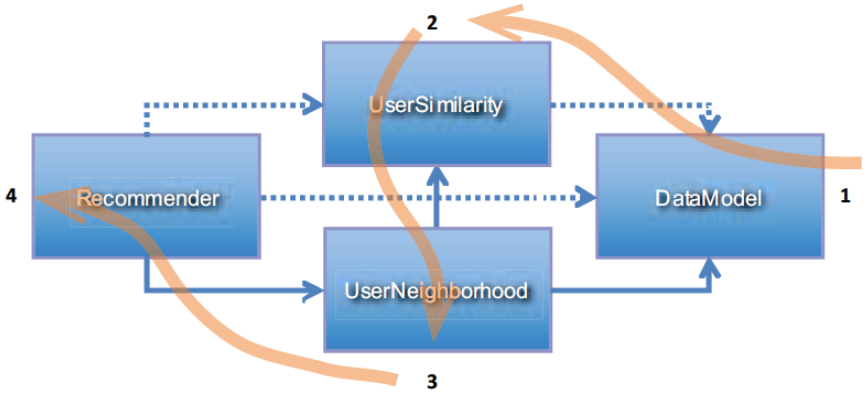
O algoritmo colaborativo básico do Mahout pode ser visualizado na Figura 8, onde *u* é o usuário alvo:

Figura 8 – Algoritmo colaborativo básico do Mahout.

```
Para cada outro usuário w
     Calcular a semelhança s entre u e w
     Obter os usuários do topo, ordenados pela similaridade,
     e chamá-los de uma vizinhança n
Para cada item i que algum usuário em n tenha atribuído uma
avaliação, mas que u não tenha atribuído uma avaliação
     Para cada outro usuário v em n que tenha atribuído uma
     avaliação para i
          Calcular a similaridade s entre u e v
          Incluir a avaliação de v atribuída a i, ponderado
          por s, em uma média móvel exponencialmente
          ponderada (EWMA)
Retornar os itens do topo, ordenados pela média ponderada
```

Fonte: adaptado de Owen *et al.* (2011, p. 44, tradução nossa)

Figura 9 – Principais classes do Mahout e seu fluxo de execução.



Fonte: Owen *et al.* (2011, p. 31)

A Figura 9 demonstra as quatro principais classes do Mahout que implementam o algoritmo apresentado anteriormente, onde pode ser observado o fluxo de execução entre elas. A seguir, é realizado um detalhamento dessas classes juntamente com uma especificação da maneira como foram utilizadas neste módulo.

DataModel: representa os dados com as preferências dos usuários. Essas preferências devem seguir um formato específico que consiste no id do usuário, id do item e o valor em números que expressa a avaliação que o usuário deu ao item. O valor com a avaliação pode ser abstraído quando for utilizada uma recomendação baseada em dados booleanos.

Esta classe trata-se de uma interface[25] que pode ser implementada por outras classes que determinam a forma como os dados com as preferências dos usuários são obtidos. As classes existentes são:

- ***FileDataModel:*** carrega os dados a partir de um arquivo no formato CSV.
- ***MySQLJDBCDataModel:*** os dados são advindos de um banco de dados MySQL.
- ***GenericDataModel*****:** é utilizada para quando se deseja popular as preferências dos usuários diretamente via código, ao invés de recorrer a uma fonte externa. Os *inputs* aceitos por esta classe utilizam outras classes da API do Mahout como a *PreferenceArray*, *FastByIDMap* e

[25] Em Java, consiste em uma classe abstrata que possui métodos abstratos que devem ser obrigatoriamente implementados, porém de forma distinta por cada classe.

FastIDSet, que são implementações específicas do Mahout para as estruturas de dados *array*, *hashmap* e *hashset*, respectivamente.

- ***GenericBooleanPrefDataModel*:** essa classe é empregada quando se deseja trabalhar com dados de preferências que não possuem avaliações, ou seja, que contenham apenas uma associação entre usuários-itens. Ela pode ser utilizada conjuntamente com as demais classes citadas.

Inicialmente, foi planejado utilizar as preferências dos usuários do Movielens salvas no banco de dados MySQL do Movie.Me. Porém, segundo Owen *et al.* (2011), o Mahout obtém uma melhor performance utilizando arquivos no formato CSV. Sendo assim, foi escolhida a classe *FileDataModel* para carregar as preferências dos usuários do Movielens do arquivo CSV. Já para as preferências do usuário alvo do Movie.Me, advindas do seu perfil do Facebook, foi empregada a classe *GenericDataModel* que utiliza um *PreferenceArray* preenchido com os dados das tabelas *users_liked_rated_movies* e *users_watched_movies* do banco de dados. Como os dados da tabela *users_watched_movies* só possuem a associação entre o usuário e os filmes marcados como assistidos por ele no Facebook, foi necessário empregar adicionalmente a classe *GenericBooleanPrefDataModel*. O mesmo não ocorre para os dados da tabela *users_liked_rated_movies* que contém avaliações expressas em notas.

UserSimilarity: é responsável por calcular a similaridade entre os usuários. Também é uma classe interface e pode ser implementada por outras classes, sendo as principais a *PearsonCorrelationSimilarity*, *EuclideanDistanceSimilary* e *LoglikelihoodSimilary*. Essas classes implementam, respectivamente, as métricas: coeficiente de correlação de Pearson, distância euclidiana e *log-likelihood*.

Para definir a métrica que seria empregada neste módulo, foram realizadas avaliações *off-line* utilizando conjuntamente o módulo de avaliação, para saber qual produzia melhores resultados. Como são utilizados dois modelos de dados – um com avaliações em formato de notas e outro com avaliações booleanas – foram empregadas duas métricas, uma para cada tipo de dado. As métricas escolhidas foram a distância euclidiana para os dados com avaliações em formato de notas e a *log-likelihood* para os dados com as avaliações booleanas.

UserNeighborhood: é responsável por definir os usuários vizinhos ao usuário alvo, ou seja, quais usuários tem gosto semelhante. Esta classe, que também é uma interface, pode ser implementada por outras duas classes, oferecendo dois modos de definir a vizinhança: com um tamanho fixo ou baseado em um determinado limiar de semelhança (*threshold*).

A classe que implementa a abordagem de tamanho fixo é a *NearestNUserNeighbourhood*. O número que determina quantos usuários semelhantes irão compor a vizinhança é geralmente arbitrário e costuma ser obtido através de experimentações. Em geral, um número pequeno significa que as recomendações serão geradas a partir de usuários muito similares ao usuário alvo, o que pode resultar em recomendações de itens já conhecidos. Por outro lado, um número muito grande pode significar o contrário, ou seja, recomendações distantes do gosto do usuário. Sendo assim, é necessário encontrar um número que represente um equilíbrio adequado ao contexto do sistema. Para o módulo de recomendação do Movie.Me foi definido o valor 200, pois foi o que apresentou melhor resultado na avaliação *off-line*.

Já a abordagem baseada em um limiar de semelhança é implementada pela classe *ThresholdUserNeighborhood* e consiste em determinar um valor mínimo que o usuário deve obter no cálculo da similaridade com o usuário alvo para poder pertencer a vizinhança. Dessa maneira, é possível assegurar que todos os usuários que pertencem a vizinhança têm um mínimo de semelhança com o usuário alvo. Apesar disso, se for estipulado um limiar muito alto, é provável que o sistema não consiga agrupar um número suficiente de usuários semelhantes, sendo necessário baixar o valor desse limiar. Por essa razão, tal abordagem não é indicada em sistemas que possuam poucos usuários. No caso do Movie.Me, foram realizados testes utilizando a classe *ThresholdUserNeighborhood*, onde o seu resultado mostrou-se superior a vizinhança de tamanho fixo para os dados com avaliações booleanos. Sendo assim, essa abordagem foi empregada pelo módulo de recomendação apenas para o cálculo que utiliza dados booleanos e utilizou um limiar de semelhança de 0.7.

Recommender: é a principal classe do componente de recomendação do Mahout e é encarregada de reunir as demais três classes citadas para produzir recomendação de itens.

As duas principais classes que implementam a interface *Recommender* são a *GenericUserBasedRecommender* e a *GenericBooleanPrefUserBasedRecommender*. Ambas são baseadas na técnica de filtragem colaborativa e se diferenciam pela primeira ser utilizada para *inputs* que possuam avaliações expressas no formato de notas e a segunda para *inputs* com avaliações booleanas.

No módulo de recomendação do Movie.Me foram empregadas ambas, ou seja, são realizados dois cálculos de recomendação separadamente. Primeiramente é efetuado o cálculo de recomendação que utiliza a classe *GenericUserBasedRecommender*, onde são utilizados os filmes curtidos e avaliados pelo usuário alvo no Facebook juntamente como as avaliações de filmes feitas pelos usuários do Movielens. Em seguida, é realizado um novo cálculo de recomendação, agora utilizando a classe *GenericBooleanPrefUserBasedRecommender*, onde são utilizados os filmes marcados como assistidos pelo usuário alvo no Facebook e as avaliações (com as notas removidas) feitas pelos usuários do Movielens. O primeiro cálculo é efetuado utilizando a métrica de distância euclidiana e a abordagem de vizinhança fixa de 200 usuários. O segundo utiliza a métrica *log-likelihood* e a vizinhança definida pelo limiar de 0.7 de semelhança. Os resultados dos dois cálculos são unidos em uma só lista que é ordenada de forma decrescente pelo peso que cada filme recomendado obteve durante o cálculo da recomendação. Além disso, essa lista também possui porcentagens distintas de filmes recomendados pelo primeiro e segundo cálculo de recomendação. Essa distinção baseia-se no seguinte princípio: se o usuário alvo possui mais filmes curtidos/avaliados do que assistidos, então são obtidos 70% dos filmes recomendados pela classe *GenericUserBasedRecommender* e 30% pela classe *GenericBooleanPrefUserBasedRecommender*. Se ocorrer o contrário, a porcentagem se inverte. No caso de possuírem o mesmo número de filmes curtidos/avaliados e assistidos, a porcentagem fica em 50%. Esses valores de porcentagens foram obtidos de forma empírica.

O motivo pelo qual não foi realizado apenas um cálculo de recomendação utilizando apenas a *GenericUserBasedRecommender* é devido ao fato de não ser possível atribuir uma nota aos filmes marcados como assistidos pelo usuário alvo, pois não há como saber se o usuário gostou ou não do filme apenas por ele ter adicionado como assistido. Isso já não ocorre com os filmes que foram curtidos, visto que o sistema tem como inferir que são seus filmes preferidos e atribuir uma

nota máxima. Sendo assim, a única abordagem viável para lidar com os filmes marcados como assistidos é utilizando a recomendação baseada em dados booleanos. O Mahout também não possibilita que sejam misturados dados booleanos e não booleanos em um mesmo cálculo de recomendação, o que também contribuiu para a decisão de realizar dois cálculos separados.

No geral, a *GenericUserBasedRecommender* produz melhores recomendações que a *GenericBooleanPrefUserBasedRecommender*. Dessa maneira, também não seria possível utilizar apenas a *GenericBooleanPrefUserBasedRecommender*, até porque descartaria desnecessariamente as notas dos filmes avaliados no Facebook pelo usuário alvo, além da sua preferência pelos filmes curtidos.

Um outro elemento que foi adicionado em ambos os cálculos de recomendação e ainda não foi comentado refere-se à incorporação da técnica de filtragem baseada em conteúdo. Isso foi realizado através da classe *IDRescorer* do Mahout. Essa classe é uma interface que pode ser implementada por outras classes customizadas que tem como objetivo aumentar o peso de determinados itens no cálculo de recomendação em função de um determinado critério. O *IDRescorer* pode ser adicionado no cálculo de recomendação através de um parâmetro no método responsável por gerar as recomendações das classes *GenericUserBasedRecommender* e *GenericBooleanPrefUserBasedRecommender*. Para o Movie.Me foram criadas seis classes customizadas que implementam a classe *IDRescorer*. Cada uma dessas classes utiliza um atributo do filme para aumentar o seu peso no cálculo de recomendação quando o mesmo corresponder a um determinado critério. Um exemplo é a classe que verifica quais gêneros de filmes são mais recorrentes nos filmes curtidos/assistidos/avaliados pelo usuário alvo no Facebook e atribui um maior peso aos filmes candidatos a serem recomendados que possuem esses gêneros.

A seguir, são descritas as classes que constituem o módulo de recomendação do Movie.Me e que utilizam as classes do Mahout apresentadas anteriormente.

- **UserBasedRecommenderWithRatedData.java (do pacote *recommender*):** é a classe que implementa a interface *Recommender* do Mahout e que possui os métodos exigidos por ela. É também a que

contém o código que constrói o cálculo de recomendação que utiliza as preferências dos usuários do Movielens com as avaliações expressas no formato de notas. Para esse fim, as classes *EuclideanDistanceSimilarity*, *NearestNUserNeighborhood* e *GenericUserBasedRecommender* do Mahout são empregadas.

- **UserBasedRecommenderWithBooleanData.java (do pacote *recommender*):** tal como a classe anterior, implementa a interface *Recommender* do Mahout e possui os métodos exigidos por ela. Além disso, contém o código que constrói o cálculo de recomendação que utiliza as preferências dos usuários do Movielens no formato booleano. Para esse propósito, emprega as classes *LogLikelihoodSimilarity*, *ThresholdUserNeighborhood* e *GenericBooleanPrefUserBasedRecommender* do Mahout.
- **UserBasedRecommenderWithUserRatedData.java (do pacote *recommender*):** é a classe que estende a classe mãe *UserBasedRecommenderWithRatedData* e adiciona ao cálculo de recomendação os filmes curtidos/avaliados no Facebook pelo usuário. Também incorpora no cálculo de recomendação a classe *MultipleRescorer*.
- **UserBasedRecommenderWithUserBooleanData.java (do pacote *recommender*):** é a classe que estende a classe mãe *UserBasedRecommenderWithBooleanData* e adiciona ao cálculo de recomendação os filmes marcados como assistidos no Facebook pelo usuário. Também incorpora no cálculo de recomendação a classe *MultipleRescorer*.
- **ActorRescorer.java (do pacote *recommender*):** é uma das classes que implementam a interface *IDRescorer* do Mahout. Tem como função verificar, para cada filme candidato a ser recomendado, se o mesmo possui algum dos atores que foram curtidos pelo usuário alvo no Facebook e, em caso positivo, aumentar o peso desse filme em 20% no cálculo de recomendação.
- **CountryRescorer.java (do pacote *recommender*):** assim como a classe anterior, implementa a interface *IDRescorer* do Mahout. Sua função é

verificar, para cada filme candidato a ser recomendado, se o mesmo pertence a um dos três países de origem mais recorrentes dos filmes curtidos/avaliados/assistidos pelo usuário no alvo no Facebook. Em caso positivo, aumenta o pesa do filme em questão em 10% no cálculo de recomendação.

- **DirectorRescorer.java (do pacote *recommender*):** é semelhante a classe *ActorRescorer* e implementa a interface *IDRescorer* do Mahout. Tem como função verificar, para cada filme candidato a ser recomendado, se o mesmo possui algum dos diretores que foram curtidos pelo usuário alvo no Facebook e, em caso positivo, aumentar o peso desse filme em 20% no cálculo de recomendação.
- **GenreRescorer.java (do pacote *recommender*):** é similar a classe *CountryRescorer* e implementa a interface *IDRescorer* do Mahout. Sua função é verificar, para cada filme candidato a ser recomendado, se o mesmo pertence a um dos quatro gêneros recorrentes dos filmes curtidos/avaliados/assistidos pelo usuário no alvo no Facebook. Em caso positivo, aumenta o pesa do filme em questão em 20% no cálculo de recomendação.
- **MovieRatingRescorer.java (do pacote *recommender*):** também implementa a interface *IDRescorer* do Mahout. Sua tarefa é verificar, para cada filme candidato a ser recomendado, se o mesmo contém a nota no IMDB maior ou igual a do filme que foi curtido/avaliado/assistido pelo usuário alvo no Facebook com a menor nota do IMDB. Os filmes que corresponderem a esse critério, têm seu peso aumentado em 10% no cálculo de recomendação.
- **MovieSimilarRescorer.java (do pacote *recommender*):** bem como as classes anteriores, implementa a interface *IDRescorer* do Mahout. Tem como tarefa verificar, para cada filme candidato a ser recomendado, se o mesmo faz parte da lista de filmes considerados similares pelo IMDB aos filmes curtidos/assistidos/avaliados pelo usuário alvo no Facebook. Se corresponder a esse critério, ocorre um aumento de 20% no peso do filme em questão no cálculo de recomendação.

- **MultipleRescorer.java (do pacote *recommender*):** essa classe, que também implementa a interface *IDRescorer* do Mahout, é responsável por unir as demais classes que implementam o *IDRescorer* em um vetor, com a finalidade de adicionar esse vetor como parâmetro no método que calcula a recomendação. Outra função importante dessa classe é de filtrar filmes candidatos a serem recomendados que, por algum motivo, não deveriam ser recomendados. Nesta classe, essa filtragem ocorre em filmes que pertencem a uma franquia e não possuem o seu filme anterior assistido pelo usuário alvo. A filtragem também ocorre para os filmes anteriores a um filme da franquia que já foi assistido pelo usuário alvo. Um exemplo que ilustra ambas situações seria: digamos que o filme “A Era do Gelo 3” esteja entre os filmes candidatos a serem recomendados. Esse filme só não será filtrado no caso de o usuário alvo ter o filme “A Era do Gelo 2” (e não ter o “A Era do Gelo 4”) entre os filmes curtidos/avaliados/assistidos. Caso contrário, ele será filtrado, visto que só faz sentido recomendar o terceiro filme da franquia, se o usuário já tenha assistido ao segundo e não tenha visto ainda o quarto.
- **MovieRecommender.java (do pacote *recommender*):** é responsável por obter os dados de avaliação dos usuários do Movielens através das classes do Mahout *FileDataModel* e *GenericBooleanPrefDataModel* e de popular o *PreferenceArray* com as preferências do usuário alvo advindas do Facebook. Em seguida, realiza a chamada das classes *UserBasedRecommenderWithUserRatedData* e *UserBasedRecommenderWithUserBooleanData* passando como parâmetro as preferências dos usuários previamente obtidas e a classe *MultipleRescorer*. Além disso, define a quantidade de filmes que serão recomendados por cada cálculo de recomendação.
- **RecommenderController.java (do pacote *controllers*):** possui métodos responsáveis por adquirir os filmes recomendados utilizando para essa finalidade a classe *MovieRecommender*.
- **recommendations.scala.html (do pacote *views*):** página onde o usuário pode visualizar uma lista com os filmes recomendados.

- **movie.scala.html (do pacote views):** é exibida quando o usuário solicita por mais detalhes do filme recomendado.

Ao final da execução do módulo de recomendação, os dados resultantes são salvos nas seguintes tabelas do banco de dados:

- **recommendations:** relaciona o id do Facebook do usuário com o id da recomendação realizada para ele.
- **recommended_movies:** relaciona o id da recomendação com os ids dos filmes que foram recomendados.

4.2.7 Módulo de avaliação

Devido ao fato do Apache Mahout também disponibilizar uma API focada na avaliação de sistemas de recomendação, o mesmo também foi escolhido para auxiliar no desenvolvimento deste módulo. Adicionalmente, foram utilizados a tecnologia AJAX e biblioteca jQuery com a finalidade de proporcionar uma melhor experiência com a interface durante o teste com usuários reais.

As classes que compõem o módulo de avaliação do Movie.Me são descritas a seguir:

- ***Evaluator.java* (do pacote *evaluator*):** contém os métodos responsáveis por executar a avaliação *off-line* utilizando diferentes métricas. A classe do Mahout que implementa a métrica MAE é a *AverageAbsoluteDifferenceRecommenderEvaluator*, para a métrica RMSE é a *RMSRecommenderEvaluator* e para as métricas de Precisão e Revocação é a *GenericRecommenderIRStatsEvaluator*. Para as métricas MAE e RMSE é necessário definir a porcentagem dos dados da base de dados que serão utilizados para treinamento e para teste, no caso do Movie.Me, foram aplicados os valores 70% e 30%, respectivamente. Já no caso da Precisão e Revocação é necessário definir um limiar (*threshold*) que determinará o que é uma recomendação boa ou ruim, o qual foi definido no Movie.Me utilizando a propriedade *GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD* do Mahout que delega ao mesmo definir esse limiar automaticamente. Além disso, também foi necessário indicar o número de *top* resultados que devem ser retornados pela precisão e revocação, que foi estabelecido

como 10, visto que o Movie.Me recomenda 10 filmes. Diferentes métricas de similaridade e definição da vizinhança foram implementados nesta classe para serem avaliados pelas três métricas de avaliação, sendo, dessa forma, possível realizar uma comparação entre eles para indicar qual obtém melhor resultado utilizando determinado tipo de dados.

- ***EvaluatorController.java* (do pacote *controllers*):** é responsável por chamar o método da classe *Evaluator* encarregado por executar a avaliação *off-line* que utiliza a métrica escolhida.
- ***evaluations.scala.hml* (do pacote *views*):** possui uma interface gráfica acessada apenas pelo usuário administrador do sistema onde são exibidos os resultados gerados por cada métrica da avaliação *off-line*.
- ***RecommenderController.*java (do pacote *recommender*):** contém métodos responsáveis por obter e salvar no banco de dados as avaliações realizadas pelos usuários reais.
- ***recommendations.scala.html* (do pacote views):** página que exibe os filmes recomendados para o usuário e permite ao usuário aprovar ou reprovar cada filme clicando em um botão.
- ***formUserOpinion.scala.html* (do pacote views):** página que possui um formulário para o usuário enviar o que achou das recomendações no geral. Esta página está incorporada a página *recommendations.scala.html* e aparece ao final, logo após a lista de filmes recomendados.

Apenas as avaliações realizadas pelos usuários reais são salvas no banco de dados, nas seguintes tabelas:

- **recommendations:** relaciona o id do Facebook do usuário, o id da recomendação realizada para ele, o id da avaliação que ele atribuiu para as recomendações e o comentário que ele possa ter realizado.
- **recommended_movies:** relaciona o id da recomendação, os ids dos filmes que foram recomendados e um valor booleano que indicar se o filme recomendado foi aprovado (1) ou reprovado (0) pelo usuário.

Os resultados obtidos pela avaliação *off-line* e teste com usuários reais utilizando este módulo são abordados no Capítulo 5.

5. ANÁLISE DOS RESULTADOS

Nesta seção, são apresentados os resultados provenientes dos diferentes métodos utilizados para avaliar o Movie.Me. A princípio, são abordados os resultados da avaliação *off-line* e, ao final, são exibidos e discutidos os resultados obtidos do teste com usuários reais.

5.1 Avaliação *off-line*

Os resultados expostos nesta seção foram adquiridos através da avaliação *off-line* e serviram como auxílio para ajuste dos parâmetros do módulo de recomendação, objetivando o emprego da melhor abordagem entre as opções disponíveis.

Primeiramente, foram avaliadas as métricas de similaridade entre usuários distância euclidiana e coeficiente de correlação de Pearson para serem utilizadas no cálculo de recomendação que utiliza como *input* as avaliações no formato de notas. Para cada métrica, foram aplicados o cálculo do MAE e RMSE considerando diferentes formas de definir a vizinhança. O Quadro 4 apresenta os valores obtidos ao se aplicar o cálculo do MAE utilizando as duas métricas de similaridade mencionadas e um tamanho de vizinhança fixo (*NearestNUserNeighborhood*) que assume diferentes valores. O valor em negrito representa o menor entre os demais e indica a melhor abordagem, visto que o MAE representa a diferença entre os valores estimados e os valores reais, então, quanto mais próximo de zero for esse valor, menor é essa diferença, sendo que o valor zero indica nenhuma diferença, ou seja, uma estimativa perfeita.

Quadro 4 – MAE utilizando o tamanho de vizinhança fixo.

Tamanho da Vizinhança	10	50	100	150	200	250	300	350	400
Distância Euclidiana	0.778	0.746	0.722	0.709	**0.702**	0.704	0.707	0.711	0.716
Coeficiente de Correlação de Pearson	0.837	0.821	0.798	0.790	0.786	0.792	0.799	0.811	0.821

Já o Quadro 5 exibe os valores obtidos ao se aplicar o cálculo do RMSE utilizando as mesmas métricas de similaridade e tamanho de vizinhança fixo.

Semelhantemente ao MAE, o melhor valor é o que for mais próximo de zero, sendo o valor em negrito o menor entre os demais.

Quadro 5 – RMSE utilizando o tamanho de vizinhança fixo.

Tamanho da Vizinhança	10	50	100	150	200	250	300	350	400
Distância Euclidiana	1.045	0.979	0.939	0.917	0.909	**0.908**	0.909	0.913	0.920
Coeficiente de Correlação de Pearson	1.117	1.058	1.023	1.018	1.016	1.027	1.039	1.059	1.076

Alternativamente, os Quadros 6 e 7 expõem os valores obtidos ao se aplicar, respectivamente, os cálculos do MAE e RMSE fazendo uso das métricas de similaridade anteriores, mas definindo o tamanho da vizinhança baseado em um limiar de semelhança entre os usuários (*ThresholdUserNeighborhood*). Como ocorre nos Quadros 4 e 5, os valores em negrito dos Quadros 6 e 7 representam o menor valor e indicam o melhor resultado.

Quadro 6 – MAE utilizando o tamanho da vizinhança baseado em um limiar de semelhança entre os usuários.

Limiar da Vizinhança	0.6	0.7	0.8	0.9
Distância Euclidiana	**0.715**	0.775	0.789	0.789
Coeficiente de Correlação de Pearson	0.794	0.811	0.822	0.836

Quadro 7 – RMSE utilizando o tamanho da vizinhança baseado em um limiar de semelhança entre os usuários

Limiar da Vizinhança	0.6	0.7	0.8	0.9
Distância Euclidiana	**0.938**	1.011	1.027	1.027
Coeficiente de Correlação de Pearson	1.019	1.041	1.058	1.073

Após a análise dos valores expostos nos quadros apresentados anteriormente, foi concluído que a melhor abordagem a ser utilizada para esse contexto é a de empregar a distância euclidiana juntamente com a definição da vizinhança em um tamanho fixo de 200 usuários, pois foi a que apresentou o menor valor entre todos.

Por outro lado, para o cálculo de recomendação com o *input* de avaliações booleanas, a literatura indica que a métrica de similaridade *log-likelihood* é a mais indicada, sendo a mesma melhor avaliada pelas métricas de precisão e revocação. Dessa forma, foram executados testes utilizando essas métricas visando definir qual tipo de definição de vizinhança é mais adequado. Os Quadros 8 e 9 apresentam os valores obtidos ao se aplicar os cálculos de precisão e revocação utilizando a métrica de similaridade *log-likelihood* com tamanho de vizinhança fixo e baseado em um limiar de semelhante entre os usuários, respectivamente. Ao contrário das métricas MAE e RMSE, os valores em negrito indicam o maior valor entre os demais, pois para as métricas de precisão e revocação, quanto maior o valor resultante, melhor é o resultado.

Quadro 8 – Precisão e revocação utilizando o tamanho de vizinhança fixo.

Tamanho da Vizinhança		**10**	**50**	**100**	**150**
Log-likelihood	**Precisão**	**0.104 (10,4%)**	0.095 (9,5%)	0.094 (9,4%)	0.097 (9,7%)
	Revocação	**0.104 (10,4%)**	0.095 (9,5%)	0.094 (9,4%)	0.097 (9,7%)

Quadro 9 – Precisão e revocação utilizando o tamanho da vizinhança baseado em um limiar de semelhança entre os usuários.

Tamanho da Vizinhança		**0.6**	**0.7**	**0.8**	**0.9**
Log-likelihood	**Precisão**	0.106 (10,6%)	**0.107 (10,7%)**	0.106(10,6%)	0.105 (10,5%)
	Revocação	0.106 (10,6%)	**0.107 (10,7%)**	0.106 (10,6%)	0.105 (10,5%)

Para esse contexto, foi constado após analisar os valores apresentados nos quadros anteriores, que o tamanho da vizinhança baseado no limiar de 0.7 usuários é a abordagem mais adequada, visto que é a que resultou o maior valor entre todos.

5.2 Teste com usuários reais

O teste com usuários reais foi efetuado com o objetivo de validar o sistema Movie.Me e avaliar a satisfação dos usuários com o conteúdo recomendado.

Para que esse teste pudesse ser realizado, foi necessário migrar o Movie.Me para a nuvem, permitindo, dessa forma, que o sistema ficasse disponível *online* aos

usuários. A empresa escolhida para esse propósito foi a Amazon AWS[26] que dispõe de um plano gratuito de 12 meses, o qual foi escolhido. Devido as limitações de *hardware* que esse plano impõe, foi necessário realizar algumas adaptações no Movie.Me visando melhorar a performance, caso contrário, o sistema demoraria alguns minutos para gerar as recomendações, o que não é considerado um tempo viável no contexto *web* atual. A primeira adaptação necessária foi a de executar uma limpeza na base de dados, eliminando todos os filmes que possuem menos de 20.000 votos no site do IMDB, visando diminuir a grande quantidade de dados presente no banco de dados. Dessa forma, a quantidade total de filmes que era inicialmente de 10.329 passou a ser de 3.628. As informações relacionadas a esses filmes removidos (atores, diretores, avaliações dos usuários do Movielens, etc.), também foram deletadas. A segunda adaptação foi a de alterar o número de usuários utilizados na definição da vizinhança do cálculo de recomendação para dados não-booleanos de 200 para 10. O motivo dessa mudança se deve ao fato de que por mais que o número 200 tenha sido o que apresentou melhor resultado na avaliação *off-line*, o incremento no número de usuários vizinhos aumenta consideravelmente o custo computacional da execução do cálculo de recomendação. Em consequência disso, foi definido o número 10 que apesar de não ser a melhor opção, possui um bom custo benefício em termos de performance e qualidade. Para o cálculo de recomendação com dados booleanos também foi adotada a vizinhança fixa de 10 usuários em detrimento da que utiliza um limiar de 0.7 de semelhança, devido aos mesmo motivos expostos anteriormente.

Realizada as devidas adaptações, o tempo para o Movie.Me gerar as recomendações foi reduzido, passando para uma média de menos de um minuto, apesar de variar conforme a quantidade de dados obtidos do Facebook do usuário alvo, ou seja, quanto maior o número de filmes curtidos/avaliados/assistidos pelo usuário, maior o tempo necessário para processar as recomendações.

O Movie.Me ficou disponível *online* para a realização dos testes durante o período de 29 de agosto de 2016 a 29 de setembro de 2016, totalizando 32 dias. Durante esse período, 23 usuários utilizaram o Movie.Me para obterem recomendações de filmes.

[26] https://aws.amazon.com/

O teste consistiu em solicitar para usuários previamente conhecidos pela autora do presente trabalho utilizarem e avaliarem o sistema Movie.Me de forma espontânea e sem supervisão. Vale ressaltar que para realizar a seleção desses usuários foi verificado primeiramente se os mesmos possuíam conta no Facebook e tinham pelo menos 1 filme curtido ou assistido ou avaliado em seu perfil na rede social. A avaliação das recomendações do Movie.Me pelo usuário foi composta por duas etapas: (1) avaliar cada um dos dez filmes recomendados clicando em um botão que aprova ou reprova a recomendação e (2) assinalar no formulário disponibilizado pelo sistema o que achou das recomendações no geral (excelente/bom/regular/ruim), com a opção de também enviar um comentário. Dos 23 usuários, apenas 1 não respondeu ao formulário e apenas 2 não avaliaram individualmente os filmes recomendados. Mais detalhes das avaliações são apresentados pelos gráficos a seguir:

Figura 10 – Gráfico com a porcentagem do que os usuários acharam das recomendações

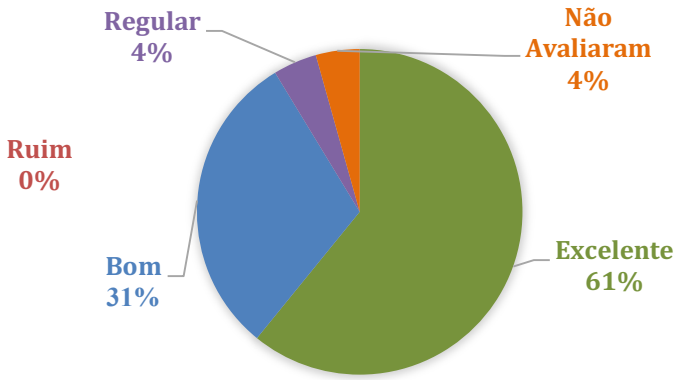


Figura 11 – Gráfico com a porcentagem da aprovação/reprovação dos filmes recomendados ao usuário

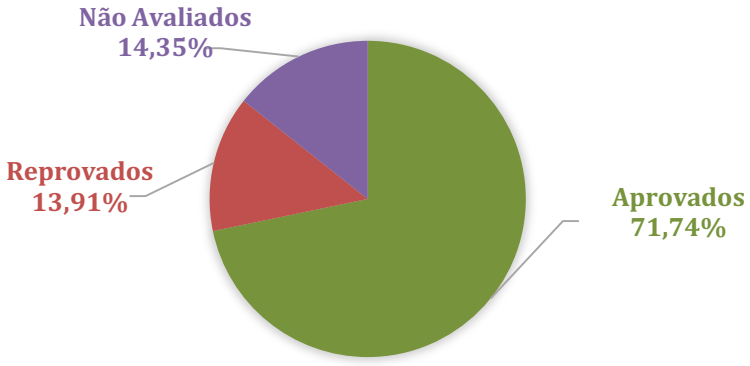
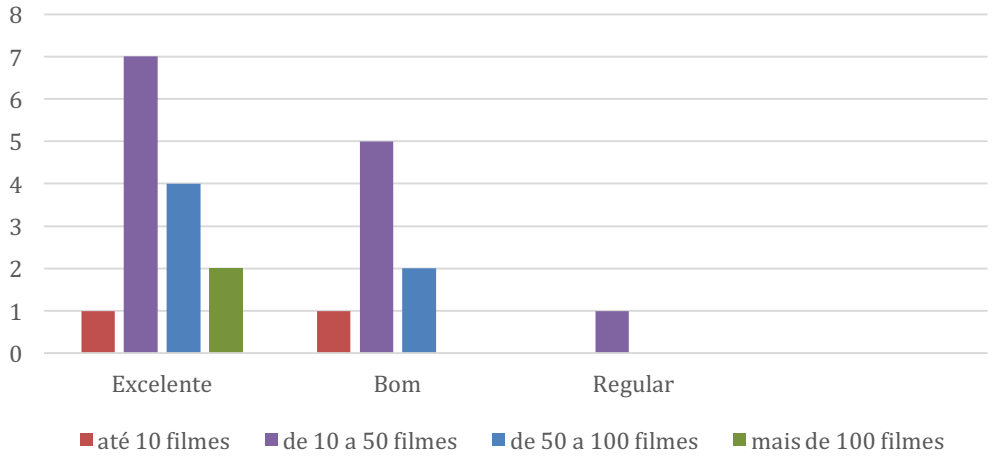


Figura 12 – Gráfico que relaciona a quantidade de filmes que o usuário curtiu/avaliou/assistiu no Facebook com a quantidade de usuários por categoria do que achou das recomendações no geral



Ao se analisar os dados presentes nos gráficos apresentados anteriormente, é concluído através da Figura 10 que o *feedback* dos usuários como um todo foi positivo e que os usuários em sua maioria aprovaram os filmes recomendados pelo Movie.Me, como demonstrado pela Figura 11. Também é verificada na Figura 12 uma relação entre a quantidade de filmes curtidos/avaliados/assistidos pelo usuário no Facebook e a sua satisfação com as recomendações, indicando que quanto maior a quantidade de dados disponíveis no perfil do usuário, maiores são as chances do Movie.Me acertar nas recomendações.

Outro aspecto que auxiliou na validação das recomendações do sistema foram os 13 comentários enviados pelos usuários via formulário. No geral, os comentários elogiaram as recomendações. Dentre os aspectos abordados nos comentários, vale ressaltar o que comentou que as recomendações não foram óbvias, apesar de interessantes, sinalizando que a técnica de recomendação colaborativa minimizou o problema de superespecialização presente em sistemas de recomendação puramente baseados em conteúdo. Além disso, usuários que tinham poucos filmes e/ou atores/diretores no Facebook, também expressaram que o sistema cumpre bem sua função mesmo com poucos dados disponíveis sobre o usuário, demonstrando sua eficácia em reduzir o *cold-start*. Um outro ponto positivo, foi o usuário que comentou que foram recomendados filmes de atores e diretores que ele aprecia, o que remete a importância de ter incorporado a recomendação baseada em conteúdo. Apesar disso, o único usuário que avaliou as recomendações

como regular, indicou um aspecto negativo que é o fato do sistema ter recomendado filmes que em sua maioria o mesmo já havia assistido e não tinha adicionado ao Facebook. Isso se caracteriza, em parte, pelo problema de esparsividade que afeta a técnica de recomendação colaborativa e acaba por dar prioridade a filmes que tem um maior número de avaliações na base de dados, ou seja, a filmes populares. Esse empecilho é minimizado pelo Movie.Me ao incorporar aspectos da filtragem baseada em conteúdo, mas, ainda assim, não é resolvido em sua totalidade, já que o sistema não tem como saber se um usuário assistiu ou não a um filme se o mesmo não foi adicionado ao seu perfil no Facebook.

O Movie.Me também não apresentou nenhuma situação que caracterizasse o problema da ovelha cinzenta (*grey sheep*), apesar de tal verificação não poder ter sido melhor analisada devido a pequena quantidade de usuários que realizaram os testes e por nenhum desses possuírem gosto demasiado atípico.

Um outro aspecto do sistema que também foi analisado durante o teste com usuários, mas que não está relacionado a avaliação realizada por eles, está na capacidade do sistema em identificar as páginas de filmes/atores/diretores do Facebook no banco de dados do Movie.Me. Foi verificado que apenas uma quantidade muito pequena de filmes e franquias contidos no Facebook dos usuários não foi identificada, o que sugere que a identificação foi executada com sucesso em sua maioria.

6. CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foram abordados conceitos acerca de sistemas de recomendação e redes sociais que serviram como base teórica para o desenvolvimento do Movie.Me, um sistema *web* de recomendação de filmes baseado na rede social Facebook. O objetivo principal do Movie.Me é o de reduzir o problema de *cold-start* ao utilizar os dados disponibilizados no perfil do Facebook do usuário alvo.

Para elaboração do sistema, foram empregadas as técnicas de filtragem colaborativa e baseada em conteúdo. Além disso, diferentes métricas para calcular a similaridade entre usuários foram testadas durante o processo de desenvolvimento do Movie.Me. Tais testes foram efetuados utilizando métricas de avaliação específicas que permitiram verificar qual a métrica de similaridade produz melhores resultados para o contexto do sistema.

Ao final do trabalho, visando avaliar se o objetivo do Movie.Me foi alcançado, foi executado um teste envolvendo usuários reais, no qual os mesmos puderam interagir com o sistema de forma *online* e validar a eficácia do conteúdo recomendado através do envio de um *feedback* explícito. Os resultados obtidos nesse teste foram satisfatórios e demonstraram que o sistema se encontra consistente quanto a finalidade para o qual foi desenvolvido. Ainda assim, algumas dificuldades foram detectadas durante esse processo, as quais são listadas a seguir:

- Algumas páginas do Facebook referentes a filmes não puderam ser reconhecidas pelo Movie.Me em sua base de dados. Isso ocorreu devido a diferentes motivos, sendo os principais o fato de o filme não constar efetivamente no banco de dados ou por a página do Facebook do filme conter em seu título palavras que não correspondem ao título do filme em si. Apesar do sistema executar uma limpeza e formatação no título da página do filme, esse processo não garante que sempre irá limpar todo conteúdo possível, pois não tem como prever todas as palavras que serão adicionadas ao título das páginas, sendo apenas as mais recorrentes consideradas.
- Em razão da base de dados de filmes utilizadas ser de origem americana, existe um número muito reduzido de filmes de origem nacional no banco de dados do Movie.Me. Dessa maneira, o sistema quase não recomenda

filmes nacionais e também falha ao reconhecer algumas das páginas do Facebook de filmes nacionais.

- Ainda que a maioria dos filmes recomendados aos usuários tenham sido aprovados por eles, foi verificado que o sistema costuma recomendar filmes considerados mais populares. Essa característica é proveniente da técnica de recomendação colaborativa, que atribui uma prioridade maior a filmes que contêm mais avaliações na base de dados. Como o Movie.Me é baseado fundamentalmente nessa técnica, esse problema também se mostrou presente no sistema.

Com a intenção de solucionar as limitações expostas anteriormente, é possível citar algumas melhorias que poderiam ser introduzidas numa segunda versão do sistema, como:

- Utilizar uma base de dados de filmes mais completa. Conforme informado no Capítulo 4 deste trabalho, o Movielens disponibiliza diversos tamanhos de base de dados, havendo o Movie.Me utilizado a menor delas. Dessa forma, seria mais indicado optar pela que possui uma maior quantidade de filmes. Adicionalmente, para lidar com a escassez de filmes nacionais, poderia ser empregado também como fonte de dados o Filmow[27] que é uma rede social brasileira de filmes e séries de televisão. Vale ressaltar, porém, que seria necessário a implementação de um *web crawler* para extrair os dados do Filmow, visto que o mesmo não possui uma API oficial para extração dos dados.
- Ao se optar por utilizar uma base de dados de filmes maior, também se torna crucial adotar uma abordagem de computação distribuída para lidar com a expressiva quantidade de dados e melhorar a performance.
- Para atribuir uma maior relevância no processo de recomendação aos filmes menos populares, é sugerido modificar o cálculo de recomendação, alterando a forma como é calculada a similaridade entre usuários. Em tal alteração, não seriam utilizadas apenas as avaliações em comum dos usuários, mas também outros aspectos que os tornam semelhantes, como idade, sexo, entre outras características.

[27] https://filmow.com/

Além das melhorias mencionadas para lidar com os desafios expostos, também é julgada interessante a implementação das seguintes funcionalidades:

- Complementarmente as páginas curtidas e filmes assistidos/avaliados, também poderiam ser extraídas outras informações do perfil do Facebook do usuário, como: postagens realizadas em sua linha do tempo, dados demográficos, seus amigos da rede, séries de televisão, livros, etc. Entretanto, para aplicar essas dados no sistema, seria necessário a implementação de uma inteligência artificial que verifique se determinada postagem que o usuário realizou é referente a um filme em questão ou se é sobre um outro assunto que compartilha do mesmo título de determinado filme. Também seria preciso avaliar de que forma os demais dados extraídos seriam empregados no cálculo de recomendação.
- Outros perfis de redes sociais do usuário alvo também poderiam ser utilizados para obter informações sobre o mesmo, como o Twitter e o Youtube, por exemplo. No entanto, a forma como o *login* e a extração simultânea dos dados de todas as redes sociais seriam efetuadas também precisaria ser estudada.

Adicionalmente, também seria pertinente executar um teste envolvendo uma quantidade maior e mais significativa de usuários reais que sejam, preferencialmente, indivíduos desconhecidos da autora deste trabalho e que podem, dessa forma, realizar uma avaliação mais imparcial.

7. REFERÊNCIAS BIBLIOGRÁFICAS

ADOMAVICIUS, Gediminas; TUZHILIN, Alexander. **Toward the next generation of recommender systems:** a survey of the state-of-the-art and possible extensions. In: IEEE Transactions on Knowledge and Data Engineering, v. 17, n. 6, p. 734-749, 2005.

BOAGLIO, Fernando. **Play framework: java para web sem servlets e com diversão**. São Paulo: Casa do Código, 2014.

BOYD, Danah M.; ELLISON, Nicole B. **Social network sites:** definition, history, and scholarship. In: Journal of Computer-Mediated Communication, v. 1, p. 210-230, 2007.

BREESE, John S.; HECKERMAN, David; KADIE, Carl. **Empirical analysis of predictive algorithms for collaborative filtering**. In: Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence, p. 43-52, 1998.

BURKE, Robin. **Hybrid recommender systems:** survey and experiments. In: User modeling and user-adapted interaction, v. 12, n. 4, p. 331–370, 2002.

BUSATTO, Cláudio José Castaldello. **O que tá valendo?** Um sistema web de recomendação de eventos. 2013. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CANHOTO, Vicente. **Recomendação de música:** comparação entre collaborative filtering e context filtering. 2013. Dissertação (Mestrado em Gestão de Sistemas de Informação) – Departamento de Ciências e Tecnologia de Informação, Instituto Universitário de Lisboa, Lisboa.

CAZELLA, Sílvio César; NUNES, Maria Augusta S. N.; REATEGUI, Eliseo Berni. **A ciência da opinião:** estado da arte em sistemas de recomendação. In: XXX Congresso da Sociedade Brasileira de Computação – Jornada de Atualização em Informática (JAI), p. 161-216, 2010.

CAZELLA, Sílvio Cézar et al. **Recomendação de objetos de aprendizagem empregando filtragem colaborativa e competências**. In: Anais do Simpósio Brasileiro de Informática na Educação, Florianópolis, 2009.

DUDEV, Minko et al. **Personalizing the search for knowledge.** In: Proc. of the 2nd International Workshop on Personalized Access, Profile Management, and Context Awareness: Databases (PersDB), 2008.

FORMIGA, Danilo de Araújo. **SuggestMe:** um sistema de recomendação utilizando web semântica para evitar o cold-start. 2014. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Departamento de Ciências Exatas, UFPB, Rio Tinto.

GOLDBERG, David et al. **Using collaborative filtering to weave an information tapestry**. In: Communications of the ACM, v. 35, n. 12, p. 61, 1992.

HERLOCKER, Jonathan L. et al. **An algorithmic framework for performing collaborative**. In: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, p-230-237, 1999.

HERLOCKER, Jonathan L. et al. **Evaluating collaborative filtering recommender systems**. In: ACM Transactions on Information Systems, v. 22, n. 1, p. 5-53, 2004.

HERLOCKER, Jonathan L.; KONSTAN, Joseph A.; RIEDL, John. **Explaining collaborative filtering recommendations.** In: ACM Conference on Computer Supported Cooperative Work, p. 241-250, 2000.

LIU, Bing. **Web data mining:** exploring hyperlinks, contents and usage data. 2 ed. Chicago: Springer, 2011.

MANNING, Christopher D.; RAGHAVAN, Prabhakar; SCHÜTZE, Hinrich. **An introduction to information retrieval**. Cambridge: Cambridge University Press, 2008.

NUNES, Maria Augusta S. N.; CAZELLA, Silvio César. **O que sua personalidade revela?** Fidelizando clientes web através de sistemas de recomendação e traços de personalidade. In: Patrícia Vilain; Valter Roesler. (Org.). Tópicos em banco de dados, multimídia e Web. 1 ed. Florianópolis: Sociedade Brasileira de Computação, v. 1, p. 91-122, 2011.

OWEN, Sean et al. **Mahout in action.** New York: Manning, 2011.

PHELAN, Owen; MCCARTHY, Kevin; SMYTH, Barry. **Using Twitter to recommend real-time topical news.** In: Proceedings of the third ACM conference on Recommender systems, 2009.

RECUERO, Raquel. **Redes sociais na internet**. Porto Alegre: Meridional, 2009.

RESNICK, Paul; VARIAN, Hal R.. **Recommender systems**. In: Communications of the ACM, v. 40, n. 3, p.56, 1997.

RICCI, Francesco et al. **Recommender systems handbook**. Springer, 2011.

SAHEBI, Shaghayegh; COHEN, William W.. **Community-based recommendations:** a solution to the cold start problem. In: Proceedings of the Workshop on Recommender Systems and the Social, Texas, 1997.

SU, Xiaoyuan; KHOSHGOFTAAR, Taghi M.. **A survey of collaborative filtering techniques.** In: Advances in Artificial Intelligence, 2009.

APÊNDICE A – ARTIGO

Movie.Me – Um Sistema de Recomendação de Filmes baseado no Facebook

Mariah Barros Cardoso

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

mariah.bcardoso@gmail.com

Abstract: *This article discusses the development of a movie recommendation system that uses the information available in the user profile of the social network Facebook to reduce the problem of cold-start. Additionally, a brief review of the literature on the area is presented, where concepts related to recommendation systems and social networks are presented. At the end, the results obtained in the test involving real users are exposed.*

Resumo: *Este artigo aborda o desenvolvimento de um sistema de recomendação de filmes que utiliza as informações disponíveis no perfil do usuário da rede social Facebook para reduzir o problema do cold-start. Adicionalmente, é realizada uma breve revisão da literatura referente a área, onde são apresentados conceitos relacionados a sistemas de recomendação e redes sociais. Ao final, são expostos os resultados obtidos no teste envolvendo usuários reais.*

1. Introdução

A rápida popularização e crescimento da Internet na última década fez dela a maior fonte de dados de acesso público existente no mundo (LIU, 2011). Devido a esse fato, o uso da Web como fonte de informação, entretenimento, cultura, produtos e serviços, tornou-se indispensável na rotina de grande parte da população. Diante desse contexto, os sistemas de recomendação surgiram com o objetivo de auxiliar o usuário a lidar com essa sobrecarga de informação, filtrando o conteúdo de forma personalizada de acordo com as preferências de cada usuário. Atualmente, são utilizados por empresas e serviços de diversos segmentos.

Apesar de bem-sucedidos, os sistemas de recomendação ainda enfrentam alguns desafios e limitações. Um dos mais conhecidos é o problema de *cold-start* que ocorre quando o sistema não possui dados suficientes sobre o usuário para realizar recomendações satisfatórias. Diversas abordagens foram propostas pela literatura para resolver esse problema, sendo uma das mais simplistas a que propõe realizar questionários prévios para obter dados explícitos do usuário. Apesar de muito utilizada, ela não é considerada ideal, pois o acréscimo de etapas extras para o uso do sistema pode acabar gerando um desinteresse por parte do usuário. Uma outra solução proposta é a de Sahebi e Cohen (1997) que sugere a utilização de sistemas externos, em especial as redes sociais, para obter informações sobre o usuário, seus interesses, afinidades e interações sociais. Essa solução é a utilizada por este trabalho que tem como objetivo principal o desenvolvimento de um sistema de recomendação de filmes que utiliza os dados disponíveis no perfil do usuário da rede social Facebook. Para atingir o objetivo, é realizado um breve estudo referente a área de recomendação da informação através de conceitos recorrentes na literatura, além de outros conceitos relacionados as demais tecnologias que compõem o sistema, como as redes sociais.

2. Sistemas de recomendação

Os sistemas de recomendação tornaram-se uma importante área de pesquisa a partir da década de 90 (ADOMAVICIUS; TUZHILIN, 2005), impulsionados principalmente pela expansão do comércio eletrônico. Atualmente, o interesse nesta área continua grande por ser rica em problemas de pesquisa e também pelo surgimento de diversos serviços que auxiliam o usuário a lidar com a sobrecarga de informação através da geração de conteúdo personalizado. (NUNES; CAZELLA, 2011).

2.1. Definição

Burke (2002) define sistema de recomendação como qualquer sistema que produza recomendações individualizadas ou que guie o usuário de forma personalizada para objetos de seu interesse dentre as diversas opções disponíveis. Também especifica que deve ser formado por três componentes: (1) dados prévios, ou seja, as informações que o sistema possui antes de iniciar as recomendações; (2) dados de entrada, que consistem nas informações previamente fornecidas pelo usuário com o propósito de auxiliar no processo de recomendação; e (3) um algoritmo que processa os dados prévios e os dados de entrada a fim de gerar recomendações.

2.2. Perfil do usuário

Cazella, Nunes e Reategui (2010) definem perfil do usuário como um conceito que reflete o interesse do usuário em relação a variados assuntos em determinado momento. Fisicamente é representado por uma base de dados que armazena informações sobre o usuário e suas preferências.

Durante a de criação do perfil do usuário, as informações podem ser coletadas de forma explícita ou implícita. Na coleta explícita, as informações são obtidas de forma direta, ou seja, o usuário insere de forma explícita informações a seu respeito e suas preferências. Já na coleta implícita, o sistema infere as preferências do usuário através das suas ações e comportamento.

A coleta explícita é considerada mais confiável e precisa, mas tem a desvantagem de depender da participação direta do usuário que nem sempre tem disponibilidade ou disposição para informar seus dados e preferências. Já a coleta implícita costuma ser mais fácil de ser obtida, apesar de ter um processamento mais difícil, pois cabe ao sistema definir se determinadas ações do usuário representam ou não suas preferências. Na prática, as duas abordagens também podem ser combinadas para se obterem melhores resultados.

2.3. Técnicas de recomendação

Os sistemas de recomendação têm sido implementados mediante o uso de diferentes técnicas que variam conforme o objetivo a ser atingido e ao tipo de item a ser recomendado.

O modelo de Burke (2002) apresenta cinco técnicas de recomendação: colaborativa, baseada em conteúdo, demográfica, baseada na utilidade e baseada no conhecimento. Dentre essas, são abordadas apenas as duas primeiras que foram as utilizadas pelo sistema deste trabalho, além de serem as mais utilizadas na prática. Além delas, também é apresentada a técnica de recomendação híbrida.

A técnica de recomendação colaborativa, também chamada de filtragem colaborativa, surgiu a partir da observação de que, no geral, os indivíduos costumam confiar em recomendações fornecidas por outros indivíduos em tomadas de decisões rotineiras. Um exemplo disso, é quando você solicita a um amigo indicações de músicas ou filmes. A

essência dessa técnica está justamente na troca de experiências entre os indivíduos que possuem interesses em comum (CAZELLA; NUNES; REATEGUI, 2010), ou seja, a ideia principal é de que usuários semelhantes costumam atribuir um feedback semelhante aos mesmos itens.

Já na técnica de recomendação baseada em conteúdo, também conhecida como filtragem baseada em conteúdo, as recomendações são realizadas utilizando o conteúdo e descrição dos itens como referência. A ideia principal dessa técnica parte da observação de que, se um usuário tem preferência sobre determinado conteúdo hoje, ele tende a manter essa preferência durante algum tempo (HERLOCKER, 2000). No caso de um usuário, por exemplo, ter avaliado de forma positiva um filme do gênero comédia, a tendência natural é de que o sistema recomende outros filmes de comédia. Da mesma maneira, se o usuário avaliou um filme que possui em seu elenco diversos atores e alguns desses atores estão presentes também em outros filmes, o sistema pode inferir que o usuário possa vir a se interessar por outros filmes envolvendo esses atores.

A técnica híbrida consiste basicamente da combinação de duas ou mais técnicas de recomendação. O objetivo dessa abordagem é justamente o de aproveitar as vantagens de uma técnica para lidar com as limitações de outra (BURKE, 2002). A grande maioria dos sistemas atuais de recomendação optam por utilizar mais de uma técnica de recomendação, sendo que a combinação mais utilizada é entre a filtragem colaborativa e a filtragem baseada em conteúdo, o que resulta em um melhor desempenho comparando-se ao uso exclusivo de uma só delas.

2.4. Desafios

Alguns dos principais desafios enfrentados pelos sistemas de recomendação são expostos por Adomavicius e Tuzhilin (2005) e Su e Khoshgoftaar (2009) e consistem em:

- ***Cold-start:*** pode ser traduzido como partida a frio e ocorre quando o sistema não possui dados iniciais suficientes sobre o usuário para que possa realizar recomendações satisfatórias. Tanto a técnica de filtragem colaborativa quanto a baseada em conteúdo sofrem com esse problema.
- **Superespecialização:** ocorre em sistemas de recomendação baseado em conteúdo e é caracterizado pela incapacidade do sistema em recomendar itens que fujam do padrão de interesses do usuário. Desse modo, ao aprender os interesses do usuário e criar um perfil para o mesmo, o sistema não inova nas recomendações realizadas devido a pouca diversidade de itens que foram previamente avaliados ou interagidos pelo usuário.
- **Análise de conteúdo limitada:** na técnica de recomendação baseada em conteúdo é necessário que o conteúdo dos itens seja computado de alguma forma, seja automaticamente ou manualmente. No entanto, para alguns domínios esse processo pode se tornar mais dificultoso, como é o caso de informações em formato de imagem, vídeo e áudio. O mesmo não ocorre na recomendação colaborativa, visto que essa técnica não utiliza o conteúdo dos itens, apenas as suas avaliações.
- **Esparsividade:** na maioria dos sistemas de recomendação colaborativos o número de avaliações efetuadas é geralmente muito menor do que o número de avaliações necessárias para se realizar a predição, ou seja, para esses sistemas é essencial a disponibilidade de uma massa crítica de usuários. Uma das consequências desse problema, é que itens que foram avaliados por poucos usuários dificilmente serão recomendados e itens que não foram avaliados nunca serão recomendados. Ou seja, a esparsividade contribui para que itens mais populares tenham mais chances de serem

recomendados que itens menos populares, o que dependendo do objetivo do sistema, pode não ser o ideal.

- ***Grey sheep:*** pode ser traduzido como problema da ovelha cinzenta e é um problema que afeta os sistemas de recomendação colaborativa e ocorre com usuários que possuem gostos atípicos ou muito diferentes dos outros usuários do sistema. Isso faz com que seja difícil encontrar usuários similares o que gera recomendações pouco eficazes. Também pode afetar sistemas de recomendação baseada em conteúdo, onde torna-se difícil de encontrar itens semelhantes aos avaliados pelo usuário alvo.

2.5. Métodos de Avaliação

Herlocker et al. (2004) propõem duas maneiras distintas para realizar a avaliação de sistemas de recomendação:

- ***Off-line:*** é realizada utilizando uma base de dados com as preferências dos usuários previamente adquiridas. Geralmente são base de dados de terceiros. O funcionamento desse método de avaliação consiste em aplicar o algoritmo em estudo para prever as preferências dos itens e posteriormente analisar os resultados utilizando uma ou mais métricas de avaliação.
- **Teste com usuários reais:** a avaliação é realizada a partir da experimentação e observação do sistema sendo utilizado por usuários reais. Esse método pode ser realizado de forma controlada ou não.

Segundo Ricci et al. (2011) diferentes métodos de avaliações podem ser utilizados conforme o estágio de desenvolvimento do sistema. Durante a fase de projeto e implementação, o ideal é o método de avaliação *off-line* para auxiliar na escolha da técnica de recomendação e algoritmos. Após o sistema ter sido concluído, o método de teste com usuários reais é o mais adequado para validar o sistema.

3. Redes socias

Recuero (2009, p. 24) descreve uma rede social como "um conjunto de dois elementos: atores (pessoas, instituições ou grupos; os nós da rede) e suas conexões (interações ou laços sociais)". Segundo Boyd e Ellison (2007) embora o propósito e nomenclatura utilizada variam de uma rede social para outra, todas apresentam uma estrutura básica que consiste em um perfil completamente ou parcialmente público do usuário onde consta seus dados pessoais e as suas conexões. O processo de inscrição geralmente é realizado mediante o preenchimento de dados em um formulário. A partir desses dados, o sistema gera um perfil associado ao usuário. O passo seguinte consiste em identificar os usuários já cadastrados no sistema e quais se deseja estabelecer uma conexão. Essas conexões podem ser tanto mútuas como unidirecionais (seguidores e fãs, por exemplo). As conexões mútuas geralmente exigem a aprovação de ambas as partes e se caracterizam por relações onde existe uma maior intimidade entre os usuários. De uma maneira geral, as redes sociais permitem que o usuário compartilhe conteúdo multimídia através do seu perfil e troque mensagens com outros usuários do sistema.

4. Movie.Me

O Movie.Me é um sistema *web* de recomendação de filmes que utiliza os dados do perfil do Facebook do usuário alvo como alternativa para reduzir o problema do *cold-start*. É baseado essencialmente na técnica de filtragem colaborativa, mas também incorpora elementos da filtragem baseada em conteúdo visando incrementar a qualidade de suas recomendações.

4.1. Fonte de dados

As fontes de dados utilizadas pelo Movie.Me são descritas a seguir:

- **Movielens:** é um sistema de recomendação colaborativo de filmes desenvolvido em 1997 pelo grupo de pesquisa Grouplens liderado por John Riedll da Universidade de Minnesota dos EUA. A sua base de dados com as avaliações dos usuários é disponibilizada para ser utilizada por outros sistemas de recomendação para fins de teste e pesquisa. A utilizada pelo Movie.Me é a do período de janeiro de 2016 e composta por 105.339 avalições de 10.329 filmes feitas por 668 usuários.
- **IMDB:** o Internet Movie Database é um portal web e base de dados online sobre filmes, séries de televisão e celebridades criado em 1990. Atualmente pertence a empresa Amazon e é um dos mais completos e populares sites do gênero. Foi utilizado pelo Movie.Me como fonte principal para se obter as informações dos filmes.
- **TMDB:** o The Movie Database surgiu em 2008 com a proposta de ser uma comunidade de compartilhamento de posters e fanart de filmes. A partir de 2009 alteraram seu foco e lançaram sua primeira base de dados de filmes. Atualmente, possui uma das maiores bases de dados sobre filmes, séries de televisão e celebridades, além de contar com uma comunidade de usuários bastante ativa. Foi utilizado pelo Movie.Me como fonte secundária para se obter os dados dos filmes.
- **Facebook:** é, hoje, a maior e mais popular rede social da Web. O Movie.Me utiliza as páginas curtidas pelo usuário e os filmes marcados como assistidos e avaliados presentes no seu perfil do Facebook.

4.2. Arquitetura e módulos

A arquitetura do Movie.Me pode ser visualizada na Figura 1.

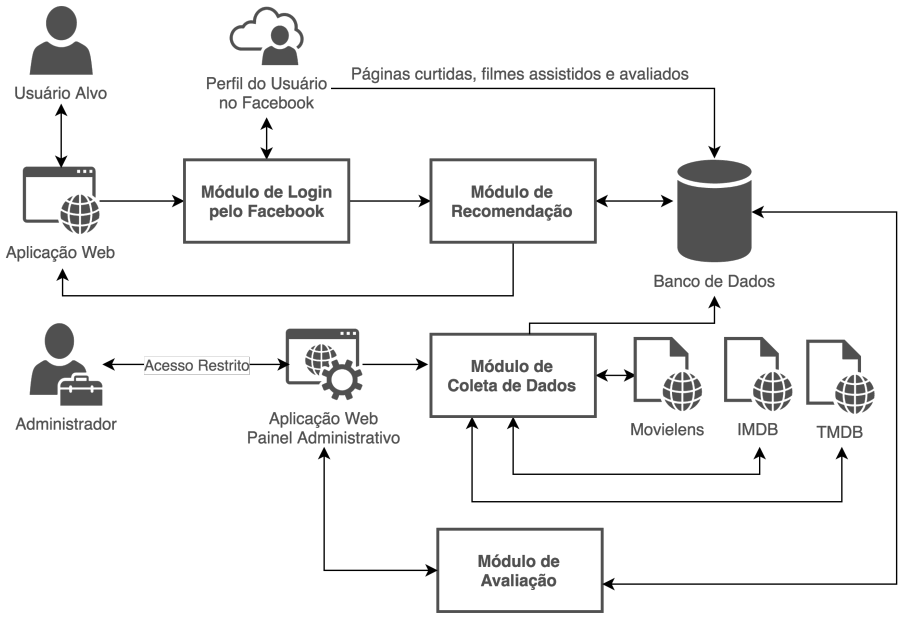


Figura 1. Arquitetura e módulos do Movie.Me

A seguir, é descrito cada módulo que compõe o sistema Movie.Me.

- **Módulo de coleta de dados:** é encarregado de popular o banco de dados do Movie.Me com as avaliações de filmes realizadas pelos usuários do sistema Movielens e para cada filme avaliado, coletar e armazenar no banco de dados suas respectivas

características (sinopse, duração, etc.) advindas das bases de dados IMDB e TMDB. O acesso a este módulo pode ser realizado apenas pelo usuário administrador do Movie.Me e sua execução deve ser efetuada antes dos demais módulos, uma vez que os dados resultantes desse processo são necessários para o funcionamento dos outros módulos que compõem o sistema.

- **Módulo de login pelo Facebook:** é responsável por logar o usuário no Movie.Me utilizando para tal a sua respectiva conta no Facebook. Sendo assim, torna-se um pré-requisito para o uso do sistema, o usuário possuir um cadastro prévio no Facebook. Se for a primeira vez que o usuário realiza o login no Movie.Me, será solicitado a sua permissão para que sejam acessados os dados disponíveis no seu perfil do Facebook. Após ter sido concedida a permissão, o módulo de login irá extrair as informações do perfil do usuário que são consideradas relevantes para o propósito do sistema e salvá-las no banco de dados. É importante ressaltar que se o usuário não autorizar o acesso aos dados, ele ficará impossibilitado de solicitar por recomendações de filmes. Caso o usuário já tenha realizado o login anteriormente, ou seja, utilizado o sistema em algum momento anterior, o módulo de login irá apenas autenticá-lo, não sendo realizada uma atualização dos dados do usuário. Isso ocorre em razão do fato de o sistema proposto neste trabalho tratar-se apenas de um protótipo, tendo sido julgado desnecessário implementar esse tipo de funcionalidade em tais circunstâncias. Uma outra tarefa que este módulo também desempenha é a de efetuar uma limpeza nos dados oriundos do Facebook, eliminando conteúdo considerado irrelevante e padronizando a formatação dos dados.
- **Módulo de Recomendação:** tem como objetivo utilizar as informações previamente coletadas pelos módulos descritos anteriormente para gerar recomendações personalizadas de filmes ao usuário do Movie.Me. A ideia principal do módulo se baseia na técnica de recomendação colaborativa e consiste em buscar por usuários do Movielens que tenham avaliações em comum com o usuário do Movie.Me a fim de utilizá-los como referência para realizar as recomendações. Além disso, também são empregadas características dos filmes como gêneros, país de origem, etc. no cálculo de recomendação, incorporando, dessa maneira, o conceito de recomendação baseada em conteúdo. O módulo também analisa as páginas curtidas no Facebook que possam ser relacionadas de forma indireta a determinados filmes na base de dados, como é o caso das páginas que representam atores, diretores e artistas em geral. Se o usuário do sistema curtiu a página de um determinado ator, por exemplo, isso também será considerado no cálculo da recomendação, que aumentará o peso dos filmes que contém o ator em questão. Este módulo só pode ser acessado por usuários logados no sistema, sendo, portanto, a execução do módulo de login pelo Facebook um pré-requisito. Após o módulo de recomendação ser executado, são exibidos ao usuário dez filmes recomendados, listados em ordem decrescente por relevância. Para cada filme exibido, é possível visualizar detalhes sobre o mesmo clicando em seu respectivo título ou pôster, o que contribui para um julgamento mais assertivo do usuário a respeito da recomendação.
- **Módulo de Avaliação:** tem como tarefas executar a avaliação *off-line* do Movie.Me e possibilitar o envio de feedbacks por parte do usuário, o que é fundamental para o teste com usuários reais. Para a avaliação *off-line*, o módulo disponibiliza diversas métricas de avaliação que são utilizadas para realizar testes no sistema com o propósito de ajustar os parâmetros do módulo de recomendação de forma que produza melhores resultados. Já para o teste com usuários reais, são disponibilizados botões para aprovar/reprovar um filme recomendado e um formulário onde o usuário pode assinalar o que achou das recomendações no geral (excelente/bom/regular/ruim) e,

opcionalmente, enviar um comentário. O acesso aos resultados da avaliação *off-line* é restrito, sendo permitido apenas para o usuário administrador do sistema. Já os botões de aprovação/reprovação e o formulário são acessíveis apenas para usuários logados no sistema e que tenham solicitado por recomendações de filmes.

5. Resultados

O teste com usuários reais foi efetuado com o objetivo de validar o sistema Movie.Me e avaliar a satisfação dos usuários com o conteúdo recomendado.

O Movie.Me ficou disponível online para a realização dos testes durante o período de 29 de agosto de 2016 a 29 de setembro de 2016, totalizando 32 dias. Durante esse período, 23 usuários utilizaram o Movie.Me para obterem recomendações de filmes.

O teste consistiu em solicitar para usuários previamente conhecidos pela autora do presente trabalho utilizarem e avaliarem o sistema Movie.Me de forma espontânea e sem supervisão. Vale ressaltar que para realizar a seleção desses usuários foi verificado primeiramente se os mesmos possuíam conta no Facebook e tinham pelo menos 1 filme curtido ou assistido ou avaliado em seu perfil na rede social. A avaliação das recomendações do Movie.Me pelo usuário foi composta por duas etapas: (1) avaliar cada um dos dez filmes recomendados clicando em um botão que aprova ou reprova a recomendação e (2) assinalar no formulário disponibilizado pelo sistema o que achou das recomendações no geral (excelente/bom/regular/ruim), com a opção de também enviar um comentário. Dos 23 usuários, apenas 1 não respondeu ao formulário e apenas 2 não avaliaram individualmente os filmes recomendados. Mais detalhes das avaliações são apresentados pelos gráficos a seguir:

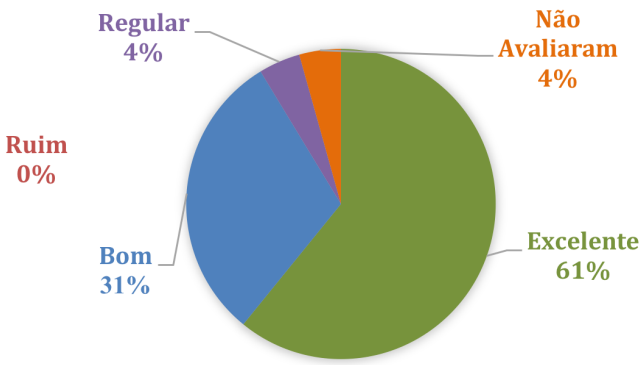


Figura 2. Gráfico com a porcentagem do que os usuários acharam das recomendações

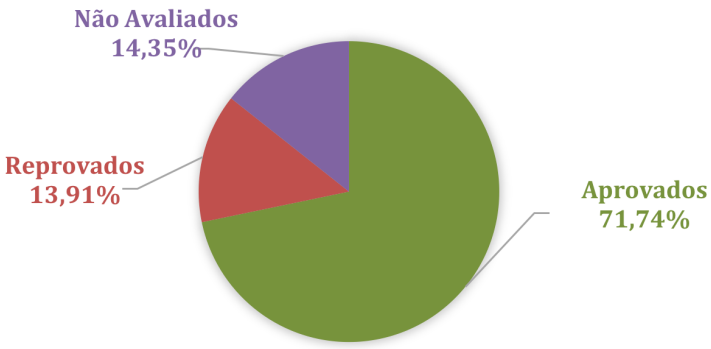


Figura 3. Gráfico com a porcentagem da aprovação/reprovação dos filmes recomendados ao usuário

Ao se analisar os dados presentes nos gráficos apresentados anteriormente, é concluído através da Figura 2 que o feedback dos usuários como um todo foi positivo e que os usuários em sua maioria aprovaram os filmes recomendados pelo Movie.Me, como demonstrado pela Figura 3.

Outro aspecto que auxiliou na validação das recomendações do sistema foram os 13 comentários enviados pelos usuários via formulário. No geral, os comentários elogiaram as recomendações. Dentre os aspectos abordados nos comentários, vale ressaltar o que comentou que as recomendações não foram óbvias, apesar de interessantes, sinalizando que a técnica de recomendação colaborativa minimizou o problema de superespecialização presente em sistemas de recomendação puramente baseados em conteúdo. Além disso, usuários que tinham poucos filmes e/ou atores/diretores no Facebook, também expressaram que o sistema cumpre bem sua função mesmo com poucos dados disponíveis sobre o usuário, demonstrando sua eficácia em reduzir o cold-start. Um outro ponto positivo, foi o usuário que comentou que foram recomendados filmes de atores e diretores que ele aprecia, o que remete a importância de ter incorporado a recomendação baseada em conteúdo. Apesar disso, o único usuário que avaliou as recomendações como regular, indicou um aspecto negativo que é o fato do sistema ter recomendado filmes que em sua maioria o mesmo já havia assistido e não tinha adicionado ao Facebook. Isso se caracteriza, em parte, pelo problema de esparsividade que afeta a técnica de recomendação colaborativa e acaba por dar prioridade a filmes que tem um maior número de avaliações na base de dados, ou seja, a filmes populares. Esse empecilho é minimizado pelo Movie.Me ao incorporar aspectos da filtragem baseada em conteúdo, mas, ainda assim, não é resolvido em sua totalidade, já que o sistema não tem como saber se um usuário assistiu ou não a um filme se o mesmo não foi adicionado ao seu perfil no Facebook.

O Movie.Me também não apresentou nenhuma situação que caracterizasse o problema da ovelha cinzenta (grey sheep), apesar de tal verificação não poder ter sido melhor analisada devido a pequena quantidade de usuários que realizaram os testes e por nenhum desses possuírem gosto demasiado atípico.

6. Conclusões e trabalhos futuros

Neste artigo foram abordados conceitos acerca de sistemas de recomendação e redes sociais que serviram como base teórica para o desenvolvimento do Movie.Me, um sistema web de recomendação de filmes baseado na rede social Facebook. O objetivo principal do Movie.Me é o de reduzir o problema de *cold-start* ao utilizar os dados disponibilizados no perfil do Facebook do usuário alvo. Para elaboração do sistema, foram empregadas as técnicas de filtragem colaborativa e baseada em conteúdo.

Ao final do trabalho, visando avaliar se o objetivo do Movie.Me foi alcançado, foi executado um teste envolvendo usuários reais, no qual os mesmos puderam interagir com o sistema de forma online e validar a eficácia do conteúdo recomendado através do envio de um feedback explícito. Os resultados obtidos nesse teste foram satisfatórios e demonstraram que o sistema se encontra consistente quanto a finalidade para o qual foi desenvolvido. Ainda assim, algumas dificuldades foram detectadas durante esse processo, as quais são listadas a seguir:

- Algumas páginas do Facebook referentes a filmes não puderam ser reconhecidas pelo Movie.Me em sua base de dados. Isso ocorreu devido a diferentes motivos, sendo os principais o fato de o filme não constar efetivamente no banco de dados ou por a página do Facebook do filme conter em seu título palavras que não correspondem ao título do filme em si. Apesar do sistema executar uma limpeza e formatação no título da página do filme, esse processo não garante que sempre irá

limpar todo conteúdo possível, pois não tem como prever todas as palavras que serão adicionadas ao título das páginas, sendo apenas as mais recorrentes consideradas.

- Em razão da base de dados de filmes utilizadas ser de origem americana, existe um número muito reduzido de filmes de origem nacional no banco de dados do Movie.Me. Dessa maneira, o sistema quase não recomenda filmes nacionais e também falha ao reconhecer algumas das páginas do Facebook de filmes nacionais.
- Ainda que a maioria dos filmes recomendados aos usuários tenham sido aprovados por eles, foi verificado que o sistema costuma recomendar filmes considerados mais populares. Essa característica é proveniente da técnica de recomendação colaborativa, que atribui uma prioridade maior a filmes que contêm mais avaliações na base de dados. Como o Movie.Me é baseado fundamentalmente nessa técnica, esse problema também se mostrou presente no sistema.

Com a intenção de solucionar as limitações expostas anteriormente, é possível citar algumas melhorias que poderiam ser introduzidas numa segunda versão do sistema, como:

- Utilizar uma base de dados de filmes mais completa. O Movielens disponibiliza diversos tamanhos de base de dados, havendo o Movie.Me utilizado a menor delas.
- Para lidar com a escassez de filmes nacionais, poderia ser empregado também uma fonte de dados nacional, como o Filmow, por exemplo.
- Ao se optar por utilizar uma base de dados de filmes maior, também se torna crucial adotar uma abordagem de computação distribuída para lidar com a expressiva quantidade de dados e melhorar a performance.
- Para atribuir uma maior relevância no processo de recomendação aos filmes menos populares, é sugerido modificar o cálculo de recomendação, alterando a forma como é calculada a similaridade entre usuários. Em tal alteração, não seriam utilizadas apenas as avaliações em comum dos usuários, mas também outros aspectos que os tornam semelhantes, como idade, sexo, entre outras características.

7. Referências

ADOMAVICIUS, Gediminas; TUZHILIN, Alexander. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. In: IEEE Transactions on Knowledge and Data Engineering, v. 17, n. 6, p. 734-749, 2005.

BOYD, Danah M.; ELLISON, Nicole B. Social network sites: definition, history, and scholarship. In: Journal of Computer-Mediated Communication, v. 1, p. 210-230, 2007.

HERLOCKER, Jonathan L. et al. Evaluating collaborative filtering recommender systems. In: ACM Transactions on Information Systems, v. 22, n. 1, p. 5-53, 2004.

HERLOCKER, Jonathan L.; KONSTAN, Joseph A.; RIEDL, John. Explaining collaborative filtering recommendations. In: ACM Conference on Computer Supported Cooperative Work, p. 241-250, 2000.

LIU, Bing. Web data mining: exploring hyperlinks, contents and usage data. 2 ed. Chicago: Springer, 2011.

NUNES, Maria Augusta S. N.; CAZELLA, Silvio César. O que sua personalidade revela? Fidelizando clientes web através de sistemas de recomendação e traços de personalidade. In: Patrícia Vilain; Valter Roesler. (Org.). Tópicos em banco de dados, multimídia e Web. 1 ed. Florianópolis: Sociedade Brasileira de Computação, v. 1, p. 91-122, 2011.

RECUERO, Raquel. Redes sociais na internet. Porto Alegre: Meridional, 2009.

RESNICK, Paul; VARIAN, Hal R.. Recommender systems. In: Communications of the ACM, v. 40, n. 3, p.56, 1997.

RICCI, Francesco et al. Recommender systems handbook. Springer, 2011.

SAHEBI, Shaghayegh; COHEN, William W.. Community-based recommendations: a solution to the cold start problem. In: Proceedings of the Workshop on Recommender Systems and the Social, Texas, 1997.

SU, Xiaoyuan; KHOSHGOFTAAR, Taghi M.. A survey of collaborative filtering techniques. In: Advances in Artificial Intelligence, 2009.

APÊNDICE B – CÓDIGO-FONTE

package controllers

MovieMe/app/controllers/AdminController.java

```
package controllers;

import java.io.IOException;
import java.net.MalformedURLException;
import java.util.List;

import org.pac4j.core.profile.CommonProfile;
import org.pac4j.oauth.profile.facebook.FacebookProfile;
import org.pac4j.play.java.JavaController;

import com.fasterxml.jackson.core.JsonProcessingException;

import crud.ActorCRUD;
import crud.CountryCRUD;
import crud.DirectorCRUD;
import crud.GenreCRUD;
import crud.MovieActorCRUD;
import crud.MovieCountryCRUD;
import crud.MovieCRUD;
import crud.MovieDirectorCRUD;
import crud.MovieGenreCRUD;
import crud.MovieSimilarCRUD;
import crud.MovielensLinkCRUD;
import crud.UserCRUD;
import models.Actor;
import models.Country;
import models.Director;
import models.Genre;
import models.Movie;
import models.MovieActor;
import models.MovieAkaJson;
import models.MovieCountry;
import models.MovieDirector;
import models.MovieGenre;
import models.MovieSimilar;
import models.MovieSimilarJson;
import models.MovielensLink;
import models.User;
import play.mvc.Result;
import services.MyApiFilmsJsonService;
import views.html.*;

// Painel de controle - acessível apenas ao administrador
public class AdminController extends JavaController {

     public static Result adminIndex() {
```

```
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
          final String urlFacebook =
getRedirectAction("FacebookClient", "/?0").getLocation();
          boolean isAdmin = false;
          boolean noData = false;
          // Verifica se usuário está logado
          if (facebookProfile == null) {
               return ok(login.render(urlFacebook));
          } else {
               // Verifica se o usuário logado é administrator
do sistema (role: (1) Administrator / (2) Usuário)
               User user =
UserCRUD.getUserByFacebookId(Long.parseLong(facebookProfile.ge
tId()));
               if (user != null) {
                    if (user.role.id == Long.valueOf(1)) {
                         isAdmin = true;
                         // Verifica se a tabela movies está
vazia
                         if (MovieCRUD.getTotalOfMovies() ==
0) {
                              noData = true;
                         }
                    }
               }
          }
          return ok(admin.render(isAdmin, noData));
     }

     public static Result importMoviesAndRelated() throws
JsonProcessingException, MalformedURLException, IOException {
          // Retorna todos os filmes da tabela
movielens_links, estipulando um intervalo
          // O intervalo é necessário devido a limitação de
requisições diárias da MyAPIFilms - 2.000 por dia
          List<MovielensLink> movielensLinks =
MovielensLinkCRUD.listAllMovielensLinksWithRange(0, 1999);

          // Para cada filme da tabela movielens_links, pega
informações no IMDB e TMDB e salva nas
          // tabelas movies e relacionadas - actors,
directors, countries e genres
          for (MovielensLink movielensLink : movielensLinks) {
               Movie movieByIdImdb =
MyApiFilmsJsonService.getMovieByIdImdb(movielensLink.id_imdb);
               // Verifica se é do tipo filme - pode ser série
de TV
               if (movieByIdImdb.json_type.equals("Movie")) {
```

```
                    // Seta o id - o mesmo id do filme no
movielens
                    movieByIdImdb.id = movielensLink.id_movie;
                    // Seta o título em inglês e/ou
alternativo
                    List<MovieAkaJson> movieAkas =
movieByIdImdb.json_akas;
                    boolean titleAlreadySettedInEn = false;
                    for (MovieAkaJson movieAkaJson :
movieAkas) {
                         // Pega o título em inglês pelo
título americano, se houver
                         if
(movieAkaJson.country.equalsIgnoreCase("USA") &&
!titleAlreadySettedInEn) {
                              movieByIdImdb.title_en =
movieAkaJson.title;
                              titleAlreadySettedInEn = true;
                              // Pega o título em inglês pelo
título britânico, se houver
                         } else if
(movieAkaJson.country.equalsIgnoreCase("UK") &&
!titleAlreadySettedInEn) {
                              movieByIdImdb.title_en =
movieAkaJson.title;
                              titleAlreadySettedInEn = true;
                         } else {
                              // Pega o título alternativo
pelo título global, se houver
                              if
(movieAkaJson.country.equalsIgnoreCase("World-wide") &&
!titleAlreadySettedInEn) {

     movieByIdImdb.title_alternative = movieAkaJson.title;
                              }
                              // Pega o título em inglês pelo
comentário, se houver
                              if (movieAkaJson.comment != null
&& movieAkaJson.comment != "") {
                                   if
(movieAkaJson.comment.equalsIgnoreCase("English title") &&
!titleAlreadySettedInEn) {
                                        movieByIdImdb.title_en
= movieAkaJson.title;
                                        titleAlreadySettedInEn
= true;
                                   }
                              }
                         }
                    }
```

```
                    // Seta o título em inglês igual ao título
original, para quando não há título
                    // em inglês americano/britânico ou no
comentário
                    // Faz isso apenas para filmes com países
de origem EUA e UK
                    if (movieByIdImdb.title_en == null) {
                         List<String> movieCountries =
movieByIdImdb.json_countries;
                         for (String movieCountry :
movieCountries) {
                              if (movieCountry.equals("USA")
|| movieCountry.equals("UK")) {
                                   movieByIdImdb.title_en =
movieByIdImdb.title_original;
                              }
                         }
                    }
                    // Seta o número de votos no IMDB
                    movieByIdImdb.votes_imdb =
Long.parseLong((movieByIdImdb.json_votes_imdb).replaceAll("[^0
-9]", ""));
                    // Seta a nota do IMDB
                    movieByIdImdb.rating_imdb =
Float.parseFloat((movieByIdImdb.json_rating_imdb).replace(",",
"."));
                    // Salva novo filme
                    MovieCRUD.saveMovie(movieByIdImdb);

                    // Salva os diretores do filme
                    for (Director director :
movieByIdImdb.json_directors) {
                         Director directorByIdImdb =
DirectorCRUD.getDirectorByIdImdb(director.id_imdb);
                         // Verifica se o diretor já existe na
base de dados
                         if (directorByIdImdb == null) {
                              // Salva novo diretor

     DirectorCRUD.saveDirector(directorByIdImdb);
                              // Salva relação filme-diretor

     MovieDirectorCRUD.saveMovieDirector(
                                        new
MovieDirector(movieByIdImdb,
DirectorCRUD.getDirectorById(directorByIdImdb.id)));
                         } else {
                              // Salva relação filme-diretor
para diretor já existente no banco de dados
```

```
     MovieDirectorCRUD.saveMovieDirector(new
MovieDirector(movieByIdImdb, directorByIdImdb));
                         }
                    }

                    // Salva os atores do filme
                    for (Actor actor :
movieByIdImdb.json_actors) {
                         Actor actorByIdImdb =
ActorCRUD.getActorByIdImdb(actor.id_imdb);
                         // Verifica se o ator já existe na
base de dados
                         if (actorByIdImdb == null) {
                              // Salva novo ator

     ActorCRUD.saveActor(actorByIdImdb);
                              // Salva relação filme-ator
                              MovieActorCRUD.saveMovieActor(
                                        new
MovieActor(movieByIdImdb,
ActorCRUD.getActorById(actorByIdImdb.id)));
                         } else {
                              // Salva relação filme-ator para
ator já existente no banco de dados

     MovieActorCRUD.saveMovieActor(new
MovieActor(movieByIdImdb, actorByIdImdb));
                         }
                    }

                    // Salva os gêneros do filme
                    for (String nameGenre :
movieByIdImdb.json_genres) {
                         Genre genreByNameEn =
GenreCRUD.getGenreByNameEn(nameGenre);
                         // Verifica se o gênero já existe na
base de dados
                         if (genreByNameEn == null) {
                              // Salva novo gênero
                              GenreCRUD.saveGenre(new
Genre(nameGenre));
                              // Salva relação filme-gênero
                              MovieGenreCRUD
                                        .saveMovieGenre(new
MovieGenre(movieByIdImdb,
GenreCRUD.getGenreByNameEn(nameGenre)));
                         } else {
                              // Salva relação filme-gênero
para gênero já existente no banco de dados
```

```
     MovieGenreCRUD.saveMovieGenre(new
MovieGenre(movieByIdImdb, genreByNameEn));
                         }
                    }

                    // Salva os países de origem do filme
                    for (String nameCountry :
movieByIdImdb.json_countries) {
                         Country countryByNameEn =
CountryCRUD.getCountryByNameEn(nameCountry);
                         // Verifica se o país de origem já
existe na base de dados
                         if (countryByNameEn == null) {
                              // Salva novo país de origem
                              CountryCRUD.saveCountry(new
Country(nameCountry));
                              // Salva relação filme-país de
origem

     MovieCountryCRUD.saveMovieCountry(
                                        new
MovieCountry(movieByIdImdb,
CountryCRUD.getCountryByNameEn(nameCountry)));
                         } else {
                              // Salva relação filme-país de
origem para país de origem já existente no banco de dados

     MovieCountryCRUD.saveMovieCountry(new
MovieCountry(movieByIdImdb, countryByNameEn));
                         }
                    }

                    // Salva os filmes indicados como
similares pelo IMDB ao filme
                    for (MovieSimilarJson movieSimilarJson :
movieByIdImdb.json_movies_similar) {
                         MovielensLink movielensLinkByIdImdb =
MovielensLinkCRUD

     .getMovielensLinkByIdImdb(movieSimilarJson.id_imdb.replac
e("tt", ""));
                         if (movielensLinkByIdImdb != null) {
                              // Salva relação filme-filme
similar

     MovieSimilarCRUD.saveMovieSimilar(new
MovieSimilar(movieByIdImdb,

     MovieCRUD.getMovieById(movielensLinkByIdImdb.id_movie)));
                         }
```

```
                    }
                }
            }
            return
redirect(routes.AdminController.adminIndex());
      }
}
```

MovieMe/app/controllers/ContentController.java

```
package controllers;

import org.pac4j.play.java.JavaController;

import play.mvc.Result;
import views.html.*;

// Páginas como informações - Sobre, Política de Privacidade,
Termos de Uso
public class ContentController extends JavaController {

      public static Result about() {
            return ok(about.render());
      }
}
```

MovieMe/app/controllers/EvaluatorController.java

```
package controllers;

import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.pac4j.core.profile.CommonProfile;
import org.pac4j.oauth.profile.facebook.FacebookProfile;
import org.pac4j.play.java.JavaController;

import crud.UserCRUD;
import models.User;
import play.mvc.Result;
import recommender.Evaluator;
import views.html.*;

// Responsável pela avaliação off-line - acessível apenas ao
administrador
public class EvaluatorController extends JavaController {

      public static Result evaluator() throws TasteException,
IOException {
```

```
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
          final String urlFacebook =
getRedirectAction("FacebookClient", "/?0").getLocation();
          boolean isAdmin = false;
          String resultMAE = "";
          String resultRMSE = "";
          String resultPrecision = "";
          String resultRecall = "";
          // Verifica se usuário está logado
          if (facebookProfile == null) {
               return ok(login.render(urlFacebook));
          } else {
               // Verifica se o usuário é administrator do
sistema (role: (1) Administrator / (2) Usuário)
               User user =
UserCRUD.getUserByFacebookId(Long.parseLong(facebookProfile.ge
tId()));
               if (user != null) {
                    if (user.role.id == Long.valueOf(1)) {
                         isAdmin = true;
                         Evaluator evaluator = new
Evaluator();
                         // MAE
                         resultMAE =
Double.toString(evaluator.getAverageAbsoluteDifferenceEvaluati
on());
                         // RMSE
                         resultRMSE =
Double.toString(evaluator.getRMSEvaluation());
                         List<Double> resultsPrecisionRecall =
evaluator.getPrecisionAndRecallEvaluation();
                         // Precisão
                         resultPrecision =
Double.toString(resultsPrecisionRecall.get(0));
                         // Revocação
                         resultRecall =
Double.toString(resultsPrecisionRecall.get(1));
                    }
               }
          }

          return ok(evaluations.render(isAdmin, resultMAE,
resultRMSE, resultPrecision, resultRecall));
     }
}
```

MovieMe/app/controllers/Global.java

```
package controllers;

import org.pac4j.core.client.Clients;
import org.pac4j.oauth.client.FacebookClient;
import org.pac4j.play.Config;

import play.Application;
import play.GlobalSettings;
import play.Play;
import play.libs.F.Promise;
import play.mvc.Http.RequestHeader;
import play.mvc.Result;
import play.mvc.Results;
import views.html.*;

public class Global extends GlobalSettings {

     @Override
     public void onStart(final Application app) {
          // Configurações necessárias para conectar o
aplicativo do facebook - Pac4j
          final String fbId =
Play.application().configuration().getString("fbId");
          final String fbSecret =
Play.application().configuration().getString("fbSecret");
          final String baseUrl =
Play.application().configuration().getString("baseUrl");

          // Configurações necessárias para funcionamento do
Pac4j
          final FacebookClient facebookClient = new
FacebookClient(fbId, fbSecret);
          // Permissões que serão solicitadas ao usuário
durante o login com o Facebook

     facebookClient.setScope("user_actions.video,user_likes");

          final Clients clients = new Clients(baseUrl +
"/callback", facebookClient);
          Config.setClients(clients);

          // Tempo de duração da sessão, em segundos
          Config.setSessionTimeout(900);
     }

     // Erro 404
     @Override
     public Promise<Result> onHandlerNotFound(RequestHeader
request) {
```

```
          return
Promise.<Result>pure(Results.notFound(error404.render()));
     }

     // Qualquer outro erro
     public Promise<Result> onError(RequestHeader request,
Throwable t) {
          return
Promise.<Result>pure(Results.internalServerError(error.render(
)));
     }
}
```

MovieMe/app/controllers/IndexController.java

```
package controllers;

import org.pac4j.core.profile.CommonProfile;
import org.pac4j.oauth.profile.facebook.FacebookProfile;
import org.pac4j.play.java.JavaController;
import org.pac4j.play.java.RequiresAuthentication;

import crud.RecommendationCRUD;
import crud.UserCRUD;
import models.User;
import play.mvc.Result;
import views.html.*;

public class IndexController extends JavaController {

     public static Result login() {
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
          final String urlFacebook =
getRedirectAction("FacebookClient", "/?0").getLocation();
          // Verifica se usuário está logado
          if (facebookProfile == null) {
               return ok(login.render(urlFacebook));
          } else {
               return
redirect(routes.IndexController.facebookIndex());
          }
     }

     @RequiresAuthentication(clientName = "FacebookClient")
     public static Result facebookIndex() {
          return protectedIndex();
     }
```

```
     private static Result protectedIndex() {
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
          User user =
UserCRUD.getUserByFacebookId(Long.parseLong(facebookProfile.ge
tId()));
          // Verifica se usuário já foi cadastrado
          if (user == null) {
               // Cadastra novo usuário
               LoginController loginService = new
LoginController(facebookProfile);
               user = loginService.registerNewUser();
          }
          // Verifica se possui dados suficientes para
realizar as recomendações
          boolean enoughData = false;
          if (user.liked_rated_movies.size() != 0 ||
user.watched_movies.size() != 0) {
               enoughData = true;
          }
          // Verifica se o sistema já gerou recomendações
anteriormente ao usuário
          boolean hasRecommendations = false;
          if
(RecommendationCRUD.getRecommendationByUserFacebookId(user.id_
user_facebook) != null) {
               hasRecommendations = true;
          }
          return ok(index.render(user.first_name, enoughData,
hasRecommendations));
     }
}
```

MovieMe/app/controllers/LoginController.java

```
package controllers;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

import org.pac4j.oauth.profile.facebook.FacebookProfile;

import com.restfb.types.Page;
```

```
import crud.ActorCRUD;
import crud.CountryCRUD;
import crud.DirectorCRUD;
import crud.GenreCRUD;
import crud.MovieCRUD;
import crud.MovieDirectorCRUD;
import crud.RoleCRUD;
import crud.UserCRUD;
import crud.UserLikeDAO;
import crud.UserLikedActorCRUD;
import crud.UserLikedDirectorCRUD;
import crud.UserLikedRatedMovieCRUD;
import crud.UserMovieCountryCRUD;
import crud.UserMovieGenreCRUD;
import crud.UserWatchedMovieCRUD;
import models.Actor;
import models.Director;
import models.Movie;
import models.MovieCountry;
import models.MovieGenre;
import models.User;
import models.UserLikedActor;
import models.UserLike;
import models.UserLikedDirector;
import models.UserLikedRatedMovie;
import models.UserMovieCountry;
import models.UserMovieGenre;
import models.UserWatchedMovie;
import services.FacebookService;

public class LoginController {

     private User user;
     private FacebookService facebookService;

     public LoginController(FacebookProfile facebookProfile) {
          // Adiciona usuário
          user = addNewUser(facebookProfile);
          facebookService = new
FacebookService(facebookProfile);
     }

     public User registerNewUser() {
          // Adiciona as páginas curtidas do usuário
          // (apenas as relacionadas a filmes/celebridades)
          addNewUserLikes();
          // Adiciona os filmes curtidos pelo usuário
          addNewUserLikedMovies();
          // Adiciona os filmes assistidos pelo usuário
          addNewUserWatchedMovies();
          // Adiciona os filmes avaliados pelo usuário
```

```
          addNewUserRatedMovies();
          // Verifica se o usuário curtiu/assistiu/avaliou
algum filme
          if (user.liked_rated_movies.size() > 0 ||
user.watched_movies.size() > 0) {
               // Adiciona os países de origem dos filmes
curtidos/assistidos/avaliados
               addNewUserMoviesCountries();
               // Adiciona os gêneros dos filmes
curtidos/assistidos/avaliados
               addNewUserMoviesGenres();
          }
          // Adiciona os atores e/ou diretores curtidos pelo
usuário
          addNewUserLikedActorsDirectors();
          return user;
     }

     private User addNewUser(FacebookProfile facebookProfile)
{
          return UserCRUD.saveUser(new
User(Long.parseLong(facebookProfile.getId()),
RoleCRUD.getRoleById(2),
                    facebookProfile.getFirstName(),
facebookProfile.getFamilyName(), 0, 0, 0, 0));
     }

     private void addNewUserLikes() {
          List<Page> likes = facebookService.getLikes();
          if (likes.size() > 0) {
               for (Page likedPage :
facebookService.getLikes()) {
                    if
(!UserLikeDAO.hasFacebookPage(user.id_user_facebook,
Long.parseLong(likedPage.getId()))) {

     user.likes.add(UserLikeDAO.saveUserLike(new
UserLike(user, Long.parseLong(likedPage.getId()),
                                        likedPage.getName(),
likedPage.getCategory())));
                    }
               }
          }
     }

     private void addNewUserLikedMovies() {
          // Retorna páginas de filmes curtidas pelo usuário
no Facebook
          List<Page> likedMovies =
facebookService.getLikedMovies();
```

```
          // Retorna páginas curtidas da categoria Personagem
Fictício
          // Necessário, pois existem páginas de filmes do
Facebook que estão com essa categoria
          List<UserLike> likedPagesCategoryFictionalCharacter
= UserCRUD.getUsersLikesByUserAndCategory(user,
                    "Fictional Character");
          if (likedPagesCategoryFictionalCharacter != null) {
               for (UserLike userLikes :
likedPagesCategoryFictionalCharacter) {
                    Page page = new Page();

     page.setId(String.valueOf(userLikes.id_page_facebook));

     page.setName(userLikes.name_page_facebook);

     page.setCategory(userLikes.category_page_facebook);
                    // Adiciona página na lista de filmes
curtidos
                    likedMovies.add(page);
               }
          }

          // Seta número total de filmes curtidos no Facebook
          // São incluídos também os que, por algum motivo,
não foram identificados pelo sistema
          // São todas páginas de filmes curtidas que o
Facebook retorna
          user.total_liked_movies = likedMovies.size();
          UserCRUD.updateUserTotalOfLikedMovies(user);

          // Identifica e adiciona os filmes curtidos,
relacionando-os com o usuário
          if (likedMovies.size() > 0) {
               for (Page moviePage : likedMovies) {
                    // Seta o ano de lançamento do filme, se
houver

     facebookService.setMoviePageReleaseData(moviePage);
                    // Normaliza título da página curtida
                    String titleMoviePage =
facebookService.normalizeFacebookMovieTitle(moviePage.getName(
));

                    // Busca pelo título original
                    List<Movie> moviesByTitleOriginal =
MovieCRUD.getMoviesByTitleOriginal(titleMoviePage);
                    if (!moviesByTitleOriginal.isEmpty()) {
                         // Foi encontrado pelo título
original
```

```
                         addLikedOrWatchedMovie(moviePage,
titleMoviePage, moviesByTitleOriginal, 1, 1);
                    } else {
                         // Busca pelo título em português
                         List<Movie> moviesByTitleBr =
MovieCRUD.getMoviesByTitleBr(titleMoviePage);
                         if (!moviesByTitleBr.isEmpty()) {
                              // Foi encontrado pelo título em
português

     addLikedOrWatchedMovie(moviePage, titleMoviePage,
moviesByTitleBr, 2, 1);
                         } else {
                              // Busca pelo título em inglês
                              List<Movie> moviesByTitleEn =
MovieCRUD.getMoviesByTitleEn(titleMoviePage);
                              if (!moviesByTitleEn.isEmpty())
{
                                   // Foi encontrado pelo
título em inglês

     addLikedOrWatchedMovie(moviePage, titleMoviePage,
moviesByTitleEn, 3, 1);
                              } else {
                                   // Busca pelo título
alternativo
                                   List<Movie>
moviesByTitleAlternative = MovieCRUD

     .getMoviesByTitleAlternative(titleMoviePage);
                                   if
(!moviesByTitleAlternative.isEmpty()) {
                                        // Foi encontrado pelo
título alternativo

     addLikedOrWatchedMovie(moviePage, titleMoviePage,
moviesByTitleAlternative, 4, 1);
                                   } else {
                                        // Busca pelo título
original da franquia, se fizer parte de uma franquia de filmes
                                        List<Movie>
moviesByTitleFranchiseOriginal = MovieCRUD

     .getMoviesByTitleFranchiseOriginal(titleMoviePage);
                                        if
(!moviesByTitleFranchiseOriginal.isEmpty()) {
                                             // Foi encontrado
pelo título original da franquia
                                             // Adiciona todos
os filmes da franquia
```

```
     addMoviesFranchise(null,
Long.parseLong(moviePage.getId()),

     moviesByTitleFranchiseOriginal, 1);
                                        } else {
                                            // Busca pelo
título em português da franquia, se fizer parte de uma
franquia de filmes
                                            List<Movie>
moviesByTitleFranchiseBr = MovieCRUD

     .getMoviesByTitleFranchiseBr(titleMoviePage);
                                            if
(!moviesByTitleFranchiseBr.isEmpty()) {
                                                // Foi
encontrado pelo título original da franquia
                                                // Adiciona
todos os filmes da franquia

     addMoviesFranchise(null,
Long.parseLong(moviePage.getId()),

     moviesByTitleFranchiseBr, 1);
                                            } else {
                                                // Busca
pelo título alternativo da franquia, se fizer parte de uma
franquia de filmes
                                                List<Movie>
moviesByTitleFranchiseAlternative = MovieCRUD

     .getMoviesByTitleFranchiseAlternative(titleMoviePage);
                                                if
(!moviesByTitleFranchiseAlternative.isEmpty()) {
                                                    // Foi
encontrado pelo título alternativo da franquia
                                                    //
Adiciona todos os filmes da franquia

     addMoviesFranchise(null,
Long.parseLong(moviePage.getId()),

     moviesByTitleFranchiseAlternative, 1);
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                }
```

```
          }
     }

     private void addNewUserWatchedMovies() {
          // Retorna filmes marcados como assistidos pelo
usuário no Facebook
          List<Page> moviesWatched =
facebookService.getWatchedMovies();

          // Seta número total de filmes marcados como
assistidos no Facebook
          // São incluídos também os que, por algum motivo,
não foram identificados pelo sistema
          // São todos os filmes marcados como assistidos que
o Facebook retorna
          user.total_watched_movies = moviesWatched.size();
          UserCRUD.updateUserTotalOfWatchedMovies(user);

          // Identifica e adiciona os filmes marcados como
assistidos, relacionando-os com o usuário
          if (moviesWatched.size() > 0) {
               for (Page moviePage : moviesWatched) {
                    // Seta o ano de lançamento do filme, se
houver

     facebookService.setMoviePageReleaseData(moviePage);
                    // Normaliza título do filme marcado como
assistido
                    String titleMoviePage =
facebookService.normalizeFacebookMovieTitle(moviePage.getName(
));

                    // Busca pelo título original
                    List<Movie> moviesByTitleOriginal =
MovieCRUD.getMoviesByTitleOriginal(titleMoviePage);
                    if (!moviesByTitleOriginal.isEmpty()) {
                         // Foi encontrado pelo título
original
                         addLikedOrWatchedMovie(moviePage,
titleMoviePage, moviesByTitleOriginal, 1, 2);
                    } else {
                         // Busca pelo título em português
                         List<Movie> moviesByTitleBr =
MovieCRUD.getMoviesByTitleBr(titleMoviePage);
                         if (!moviesByTitleBr.isEmpty()) {
                              // Foi encontrado pelo título em
português

     addLikedOrWatchedMovie(moviePage, titleMoviePage,
moviesByTitleBr, 2, 2);
                         } else {
```

```
                                // Busca pelo título em inglês
                                List<Movie> moviesByTitleEn =
MovieCRUD.getMoviesByTitleEn(titleMoviePage);
                                if (!moviesByTitleEn.isEmpty())
{
                                     // Foi encontrado pelo
título em inglês

     addLikedOrWatchedMovie(moviePage, titleMoviePage,
moviesByTitleEn, 3, 2);
                                } else {
                                     // Busca pelo título
alternativo
                                     List<Movie>
moviesByTitleAlternative = MovieCRUD

     .getMoviesByTitleAlternative(titleMoviePage);
                                     if
(!moviesByTitleAlternative.isEmpty()) {
                                          // Foi encontrado pelo
título alternativo

     addLikedOrWatchedMovie(moviePage, titleMoviePage,
moviesByTitleAlternative, 4, 2);
                                     } else {
                                          // Busca pelo título
original da franquia, se fizer parte de uma franquia de filmes
                                          List<Movie>
moviesByTitleFranchiseOriginal = MovieCRUD

     .getMoviesByTitleFranchiseOriginal(titleMoviePage);
                                          if
(!moviesByTitleFranchiseOriginal.isEmpty()) {
                                               // Foi encontrado
pelo título original da franquia
                                               // Adiciona todos
os filmes da franquia

     addMoviesFranchise(null,
Long.parseLong(moviePage.getId()),

     moviesByTitleFranchiseOriginal, 2);
                                          } else {
                                               // Busca pelo
título em português da franquia, se fizer parte de uma
franquia de filmes
                                               List<Movie>
moviesByTitleFranchiseBr = MovieCRUD

     .getMoviesByTitleFranchiseBr(titleMoviePage);
```

```
                                                  if
(!moviesByTitleFranchiseBr.isEmpty()) {
                                                       // Foi
encontrado pelo título em português da franquia
                                                       // Adiciona
todos os filmes da franquia

     addMoviesFranchise(null,
Long.parseLong(moviePage.getId()),

     moviesByTitleFranchiseBr, 2);
                                                  } else {
                                                       // Busca
pelo título alternativo da franquia, se fizer parte de uma
franquia de filmes
                                                       List<Movie>
moviesByTitleFranchiseAlternative = MovieCRUD

     .getMoviesByTitleFranchiseAlternative(titleMoviePage);
                                                       if
(!moviesByTitleFranchiseAlternative.isEmpty()) {
                                                            // Foi
encontrado pelo título alternativo da franquia
                                                            //
Adiciona todos os filmes da franquia

     addMoviesFranchise(null,
Long.parseLong(moviePage.getId()),

     moviesByTitleFranchiseAlternative, 2);
                                                       }
                                                  }
                                             }
                                        }
                                   }
                              }
                         }
                    }
               }
          }
     }

     private void addNewUserRatedMovies() {
          Map<Movie, Long> moviesLikedAndWatched = new
HashMap<Movie, Long>();

          // Pega os filmes curtidos pelo usuário no Facebook
- já foram identificados e adicionados
          if (user.liked_rated_movies.size() > 0) {
               for (UserLikedRatedMovie userLikedRatedMovie :
user.liked_rated_movies) {
```

```
     moviesLikedAndWatched.put(userLikedRatedMovie.movie,
userLikedRatedMovie.id_page_facebook);
                }
          }

          // Pega os filmes marcados como assistidos pelo
usuário no Facebook - já foram identificados e adicionados
          if (user.watched_movies.size() > 0) {
               for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {

     moviesLikedAndWatched.put(userWatchedMovie.movie,
userWatchedMovie.id_page_facebook);
                }
          }

          // Retorna filmes avaliados pelo usuário no Facebook
com sua respectiva nota
          Map<Page, Integer> moviesRated =
facebookService.getRatedMovies();

          // Seta número total de filmes avaliados pelo
usuário no Facebook
          // São incluídos também os que, por algum motivo,
não foram identificados pelo sistema
          // São todos os filmes avaliados pelo usuário que o
Facebook retorna
          user.total_rated_movies = moviesRated.size();
          UserCRUD.updateUserTotalOfRatedMovies(user);

          // Identifica e adiciona os filmes avaliados,
relacionando-os com o usuário
          // Utiliza para isso os filmes curtidos e
assistidos, já identificados e adicionados anteriormente
          if (moviesLikedAndWatched.size() > 0 &&
moviesRated.size() > 0) {
               for (Map.Entry<Page, Integer> movieRated :
moviesRated.entrySet()) {
                    for (Map.Entry<Movie, Long>
movieAndPageFacebookId : moviesLikedAndWatched.entrySet()) {
                         if
((Long.parseLong(movieRated.getKey().getId()) ==
movieAndPageFacebookId.getValue())) {
                              if
(UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,

     movieAndPageFacebookId.getKey().id)) {
                                   UserLikedRatedMovie
userLikedRatedMovie = UserLikedRatedMovieCRUD
```

```
     .getUserLikedRatedMovieByUserFacebookIdAndMovieId(user.id
_user_facebook,

     movieAndPageFacebookId.getKey().id);
                                        userLikedRatedMovie.rating
= movieRated.getValue().floatValue();

     UserLikedRatedMovieCRUD.updateUserLikedRatedMovieRating(u
serLikedRatedMovie);
                                } else {

     user.liked_rated_movies.add(UserLikedRatedMovieCRUD.saveU
serLikedRatedMovie(
                                                      new
UserLikedRatedMovie(user, movieAndPageFacebookId.getKey(),

     movieAndPageFacebookId.getValue(),
movieRated.getValue().floatValue())));
                                }
                           }
                      }
                 }
            }
     }

     private void addNewUserMoviesCountries() {
          // Adiciona os países de origem dos filmes
curtidos/assistidos/avaliados pelo usuário no Facebook
          // Utiliza para isso os filmes
curtidos/assistidos/avaliados já identificados e adicionados
anteriormente
          // Também computa a incidência de cada país de
origem

          List<MovieCountry> moviesLikedOrWatchedCountries =
new ArrayList<MovieCountry>();
          if (user.liked_rated_movies.size() > 0) {
               for (UserLikedRatedMovie userLikedRatedMovie :
user.liked_rated_movies) {

     moviesLikedOrWatchedCountries.addAll(userLikedRatedMovie.
movie.countries);
               }
          }
          if (user.watched_movies.size() > 0) {
               for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {

     moviesLikedOrWatchedCountries.addAll(userWatchedMovie.mov
ie.countries);
```

```
                }
            }

            List<Long> userMoviesIdsCountries = new
ArrayList<Long>();
            for (MovieCountry movieCountry :
moviesLikedOrWatchedCountries) {

      userMoviesIdsCountries.add(movieCountry.country.id);
            }

            Map<Long, Integer> mapCountries = new HashMap<Long,
Integer>();
            for (Long temp : userMoviesIdsCountries) {
                 Integer count = mapCountries.get(temp);
                 mapCountries.put(temp, (count == null) ? 1 :
count + 1);
            }

            Map<Long, Integer> treeMapCountries = new
TreeMap<Long, Integer>(mapCountries);
            for (Map.Entry<Long, Integer> entry :
treeMapCountries.entrySet()) {

      user.movies_countries.add(UserMovieCountryCRUD.saveUserMo
vieCountry(
                           new UserMovieCountry(user,
CountryCRUD.getCountryById(entry.getKey()),
entry.getValue())));
            }
      }

      private void addNewUserMoviesGenres() {
            // Adiciona os gêneros dos filmes
curtidos/assistidos/avaliados pelo usuário no Facebook
            // Utiliza para isso os filmes
curtidos/assistidos/avaliados já identificados e adicionados
anteriormente
            // Também computa a incidência de cada gênero

            List<MovieGenre> moviesLikedOrWatchedGenres = new
ArrayList<MovieGenre>();
            if (user.liked_rated_movies.size() > 0) {
                  for (UserLikedRatedMovie userLikedRatedMovie :
user.liked_rated_movies) {

      moviesLikedOrWatchedGenres.addAll(userLikedRatedMovie.mov
ie.genres);
                  }
            }
            if (user.watched_movies.size() > 0) {
```

```
                for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {

     moviesLikedOrWatchedGenres.addAll(userWatchedMovie.movie.
genres);
                }
          }

          List<Long> userMoviesNamesGenres = new
ArrayList<Long>();
          for (MovieGenre movieGenre :
moviesLikedOrWatchedGenres) {
                userMoviesNamesGenres.add(movieGenre.genre.id);
          }

          Map<Long, Integer> mapGenres = new HashMap<Long,
Integer>();
          for (Long temp : userMoviesNamesGenres) {
                Integer count = mapGenres.get(temp);
                mapGenres.put(temp, (count == null) ? 1 : count
+ 1);
          }

          Map<Long, Integer> treeMapGenres = new TreeMap<Long,
Integer>(mapGenres);
          for (Map.Entry<Long, Integer> entry :
treeMapGenres.entrySet()) {

     user.movies_genres.add(UserMovieGenreCRUD.saveUserMovieGe
nre(
                           new UserMovieGenre(user,
GenreCRUD.getGenreById(entry.getKey()), entry.getValue())));
          }
     }

     private void addNewUserLikedActorsDirectors() {
          // Retorna as páginas de atores e/ou diretores
curtidas pelo usuário no Facebook
          List<Page> actorsDirectorsLiked =
facebookService.getLikedActorsAndDirectors();

          // Seta número total de atores e diretores curtidos
pelo usuário no Facebook
          // São incluídos também os que, por algum motivo,
não foram identificados pelo sistema
          // São todos os atores e diretores curtidos pelo
usuário que o Facebook retorna
          user.total_liked_actors_directors =
actorsDirectorsLiked.size();

     UserCRUD.updateUserTotalOfLikedActorsDirectors(user);
```

```
          // Identifica e adiciona os atores e diretores
curtidos, relacionando-os com o usuário
          // A identificação é realizada pelo nome do
ator/diretor
          Map<Actor, Long> actorsByName = new HashMap<Actor,
Long>();
          Map<Director, Long> directorsByName = new
HashMap<Director, Long>();
          if (actorsDirectorsLiked.size() > 0) {
               for (Page actorDirectorPage :
actorsDirectorsLiked) {
                    String nameActorDirector = facebookService

     .normalizeFacebookActorDirectorName(actorDirectorPage.get
Name());
                    for (Actor actor :
ActorCRUD.getActorsByName(nameActorDirector)) {
                         if (!actorsByName.containsKey(actor))
{
                              actorsByName.put(actor,
Long.parseLong(actorDirectorPage.getId()));
                         }
                    }
                    for (Director director :
DirectorCRUD.getDirectorsByName(nameActorDirector)) {
                         if
(!directorsByName.containsKey(director)) {
                              directorsByName.put(director,
Long.parseLong(actorDirectorPage.getId()));
                         }
                    }
               }
               if (!actorsByName.isEmpty()) {
                    for (Map.Entry<Actor, Long> actor :
actorsByName.entrySet()) {
                         if
(!UserLikedActorCRUD.hasActor(user.id_user_facebook,
actor.getKey().id)) {

     user.liked_actors.add(UserLikedActorCRUD

     .saveUserLikedActor(new UserLikedActor(user,
actor.getKey(), actor.getValue())));
                         }
                    }
               }
               if (!directorsByName.isEmpty()) {
                    for (Map.Entry<Director, Long> director :
directorsByName.entrySet()) {
```

```
                        if
(!UserLikedDirectorCRUD.hasDirector(user.id_user_facebook,
director.getKey().id)) {

     user.liked_directors.add(UserLikedDirectorCRUD.saveUserLi
kedDirector(
                                          new
UserLikedDirector(user, director.getKey(),
director.getValue())));
                        }
                    }
               }
          }
     }

     private void addLikedOrWatchedMovie(Page moviePage,
String titleMoviePage, List<Movie> moviesByTitle,
               int typeOfTitle, int likedOrWatched) {
          // Realiza a identificação de um filme
curtido/assistido/avaliado pelo usuário no Facebook
          // A identificação é feita buscando pelo título do
filme em diferentes idiomas e pelo título da franquia, caso
fizer parte de uma
          // Também é utilizado o ano de lançamento do filme e
o diretor, quando essas informações estiverem disponíveis na
página do filme

          // typeOfTitle - (1) original / (2) em português /
(3) em inglês / (4) alternativo
          // likedOrWatched - (1) curtido / (2) assistido

          boolean foundMovieByTitleAndDateDirector = false;
          // Página do filme tem o atributo de data de
lançamento e/ou diretor
          if (moviePage.getReleaseDate() != null ||
moviePage.getDirectedBy() != null) {
               // Página do filme tem só o ano de lançamento
               if (moviePage.getReleaseDate() != null &&
moviePage.getDirectedBy() == null) {
                    Movie movieByTitleAndYear = null;
                    // Busca pelo título original e ano de
lançamento
                    if (typeOfTitle == 1) {
                         movieByTitleAndYear =
MovieCRUD.getMovieByTitleOriginalAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
                    // Busca pelo título em português e
ano de lançamento
                    } else if (typeOfTitle == 2) {
```

```
                        movieByTitleAndYear =
MovieCRUD.getMovieByTitleBrAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
                        // Busca pelo título em inglês e ano
de lançamento
                    } else if (typeOfTitle == 3) {
                        movieByTitleAndYear =
MovieCRUD.getMovieByTitleEnAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
                        // Busca pelo título alternativo e
ano de lançamento
                    } else if (typeOfTitle == 4) {
                        movieByTitleAndYear =
MovieCRUD.getMovieByTitleAlternativeAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
                    }
                    if (movieByTitleAndYear != null) {
                        // Verifica se o filme faz parte de
uma franquia, é o primeiro da franquia e se o seu título é
igual ao título da franquia
                        if
(isMovieFranchise(movieByTitleAndYear, typeOfTitle)) {

     addMoviesFranchise(movieByTitleAndYear,
Long.parseLong(moviePage.getId()), null,
                                            likedOrWatched);
                        } else {
                            // Se o filme já foi salvo como
curtido, não será salvo novamente como curtido ou assistido
                            if
(!UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,
movieByTitleAndYear.id)) {
                                if (likedOrWatched == 1) {
                                    // Salva filme curtido
e relaciona-o ao usuário

     user.liked_rated_movies.add(UserLikedRatedMovieCRUD

     .saveUserLikedRatedMovie(new UserLikedRatedMovie(user,
movieByTitleAndYear,

     Long.parseLong(moviePage.getId()), Float.valueOf(5))));
                                } else {
                                    // Se o filme já foi
salvo como assistido, não será salvo novamente como curtido ou
assistido
```

```
                                        if
(!UserWatchedMovieCRUD.hasMovie(user.id_user_facebook,
movieByTitleAndYear.id)) {
                                             // Salva filme
assistido e relaciona-o ao usuário

     user.watched_movies

     .add(UserWatchedMovieCRUD.saveUserWatchedMovie(new
UserWatchedMovie(user,

     movieByTitleAndYear,
Long.parseLong(moviePage.getId()))));
                                        }
                                   }
                              }
                         }
                         // Filme encontrado
                         foundMovieByTitleAndDateDirector =
true;
                    }
                    // Página do filme tem só o diretor
               } else if (moviePage.getReleaseDate() == null
&& moviePage.getDirectedBy() != null) {
                    // Busca filme pelo título e diretor
                    Movie movieByTitleAndDirector = null;
                    for (Movie movie : moviesByTitle) {
                         for (Director director :
MovieDirectorCRUD.getDirectorsByMovieId(movie.id)) {
                              if
(moviePage.getDirectedBy().trim().equalsIgnoreCase(director.na
me)) {
                                   movieByTitleAndDirector =
movie;
                                   break;
                              }
                         }
                    }
                    if (movieByTitleAndDirector != null) {
                         // Verifica se o filme faz parte de
uma franquia, é o primeiro da franquia e se o seu título é
igual ao título da franquia
                         if
(isMovieFranchise(movieByTitleAndDirector, typeOfTitle)) {

     addMoviesFranchise(movieByTitleAndDirector,
Long.parseLong(moviePage.getId()), null,
                                   likedOrWatched);
                         } else {
                              // Se o filme já foi salvo como
curtido, não será salvo novamente como curtido ou assistido
```

```
                                if
(!UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,
movieByTitleAndDirector.id)) {
                                     if (likedOrWatched == 1) {
                                          // Salva filme curtido
e relaciona-o ao usuário

     user.liked_rated_movies.add(UserLikedRatedMovieCRUD

     .saveUserLikedRatedMovie(new UserLikedRatedMovie(user,
movieByTitleAndDirector,

     Long.parseLong(moviePage.getId()), Float.valueOf(5))));
                                     } else {
                                          // Se o filme já foi
salvo como assistido, não será salvo novamente como curtido ou
assistido
                                          if
(!UserWatchedMovieCRUD.hasMovie(user.id_user_facebook,
movieByTitleAndDirector.id)) {
                                               // Salva filme
assistido e relaciona-o ao usuário

     user.watched_movies

     .add(UserWatchedMovieCRUD.saveUserWatchedMovie(new
UserWatchedMovie(user,

     movieByTitleAndDirector,
Long.parseLong(moviePage.getId()))));
                                          }
                                     }
                                }
                           }
                           // Filme encontrado
                           foundMovieByTitleAndDateDirector =
true;
                      }
                 } else {
                      // Página do filme tem ano de lançamento e
diretor
                      Movie movieByTitleAndYearAndDirector =
null;
                      Movie movieByTitleAndYear = null;
                      // Busca pelo título original e ano de
lançamento
                      if (typeOfTitle == 1) {
                           movieByTitleAndYear =
MovieCRUD.getMovieByTitleOriginalAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
```

```
                        // Busca pelo título em português e
ano de lançamento
                    } else if (typeOfTitle == 2) {
                        movieByTitleAndYear =
MovieCRUD.getMovieByTitleBrAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
                        // Busca pelo título em inglês e ano
de lançamento
                    } else if (typeOfTitle == 3) {
                        movieByTitleAndYear =
MovieCRUD.getMovieByTitleEnAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
                        // Busca pelo título alternativo e
ano de lançamento
                    } else if (typeOfTitle == 4) {
                        movieByTitleAndYear =
MovieCRUD.getMovieByTitleAlternativeAndYear(titleMoviePage,

     Integer.parseInt(moviePage.getReleaseDate()));
                    }
                    // Busca pelo título, ano e diretor
                    if (movieByTitleAndYear != null) {
                        for (Director director :
MovieDirectorCRUD.getDirectorsByMovieId(movieByTitleAndYear.id
)) {
                            if
(moviePage.getDirectedBy().trim().equalsIgnoreCase(director.na
me)) {

     movieByTitleAndYearAndDirector = movieByTitleAndYear;
                                break;
                            }
                        }
                        if (movieByTitleAndYearAndDirector !=
null) {
                            // Verifica se o filme faz parte
de uma franquia, é o primeiro da franquia e se o seu título é
igual ao título da franquia
                            if
(isMovieFranchise(movieByTitleAndYearAndDirector,
typeOfTitle)) {

     addMoviesFranchise(movieByTitleAndYearAndDirector,
Long.parseLong(moviePage.getId()), null,
                                    likedOrWatched);
                            } else {
                                // Se o filme já foi salvo
como curtido, não será salvo novamente como curtido ou
assistido
```

```
                                        if
(!UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,

     movieByTitleAndYearAndDirector.id)) {
                                             if (likedOrWatched ==
1) {
                                                  // Salva filme
curtido e relaciona-o ao usuário

     user.liked_rated_movies.add(UserLikedRatedMovieCRUD.saveU
serLikedRatedMovie(
                                                                 new
UserLikedRatedMovie(user, movieByTitleAndYearAndDirector,

     Long.parseLong(moviePage.getId()), Float.valueOf(5))));
                                             } else {
                                                  // Se o filme já
foi salvo como assistido, não será salvo novamente como
curtido ou assistido
                                                  if
(!UserWatchedMovieCRUD.hasMovie(user.id_user_facebook,

     movieByTitleAndYearAndDirector.id)) {
                                                                 // Salva
filme assistido e relaciona-o ao usuário

     user.watched_movies.add(UserWatchedMovieCRUD.saveUserWatc
hedMovie(

     new UserWatchedMovie(user,
movieByTitleAndYearAndDirector,

          Long.parseLong(moviePage.getId())));
                                                  }
                                             }
                                        }
                                   }
                                   // Filme encontrado
                                   foundMovieByTitleAndDateDirector
= true;
                              }
                         }
                    }
          }
          // A página do filme não tem ano de lançamento e
diretor ou os mesmo não foram identificados anteriormente
          if (foundMovieByTitleAndDateDirector == false) {
               for (Movie movie : moviesByTitle) {
                    // Verifica se o filme faz parte de uma
franquia, é o primeiro da franquia e se o seu título é igual
ao título da franquia
```

```
                    if (isMovieFranchise(movie, typeOfTitle))
{
                         addMoviesFranchise(movie,
Long.parseLong(moviePage.getId()), null, likedOrWatched);
                    } else {
                         // Se o filme já foi salvo como
curtido, não será salvo novamente como curtido ou assistido
                         if
(!UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,
movie.id)) {
                              if (likedOrWatched == 1) {
                                   // Salva filme curtido e
relaciona-o ao usuário
                                   user.liked_rated_movies

     .add(UserLikedRatedMovieCRUD.saveUserLikedRatedMovie(new
UserLikedRatedMovie(user,
                                                             movie,
Long.parseLong(moviePage.getId()), Float.valueOf(5))));
                              } else {
                                   // Se o filme já foi salvo
como assistido, não será salvo novamente como curtido ou
assistido
                                   if
(!UserWatchedMovieCRUD.hasMovie(user.id_user_facebook,
movie.id)) {
                                        // Salva filme
assistido e relaciona-o ao usuário

     user.watched_movies.add(UserWatchedMovieCRUD.saveUserWatc
hedMovie(
                                                       new
UserWatchedMovie(user, movie,
Long.parseLong(moviePage.getId()))));
                                   }
                              }
                         }
                    }
               }
          }
     }

     private boolean isMovieFranchise(Movie movie, int
typeOfTitle) {
          // Verifica se o filme faz parte de uma franquia, é
o primeiro da franquia e se o seu título é igual ao título da
franquia

          // Busca pelo título original do filme e pelo título
original/em português e alternativo da franquia
          if (typeOfTitle == 1) {
```

```
                if (movie.is_movie_franchise == 1 &&
movie.order_franchise == 1
                        &&
(movie.title_original.equalsIgnoreCase(movie.title_franchise_o
riginal)
                                    ||
movie.title_original.equalsIgnoreCase(movie.title_franchise_br
)
                                    ||
movie.title_original.equalsIgnoreCase(movie.title_franchise_al
ternative))) {
                    return true;

                } else {
                    return false;
                }
                // Busca pelo título em português do filme e
pelo título original/em português e alternativo da franquia
           } else if (typeOfTitle == 2) {
                if (movie.is_movie_franchise == 1 &&
movie.order_franchise == 1
                        &&
(movie.title_br.equalsIgnoreCase(movie.title_franchise_origina
l)
                                    ||
movie.title_br.equalsIgnoreCase(movie.title_franchise_br)
                                    ||
movie.title_br.equalsIgnoreCase(movie.title_franchise_alternat
ive))) {
                    return true;

                } else {
                    return false;
                }
                // Busca pelo título em inglês do filme e pelo
título original/em português e alternativo da franquia
           } else if (typeOfTitle == 3) {
                if (movie.is_movie_franchise == 1 &&
movie.order_franchise == 1
                        &&
(movie.title_en.equalsIgnoreCase(movie.title_franchise_origina
l)
                                    ||
movie.title_en.equalsIgnoreCase(movie.title_franchise_br)
                                    ||
movie.title_en.equalsIgnoreCase(movie.title_franchise_alternat
ive))) {
                    return true;

                } else {
                    return false;
```

```
                }
                // Busca pelo título alternativo do filme e
pelo título original/em português e alternativo da franquia
          } else if (typeOfTitle == 4) {
                if (movie.is_movie_franchise == 1 &&
movie.order_franchise == 1
                          &&
(movie.title_alternative.equalsIgnoreCase(movie.title_franchis
e_original)
                                    ||
movie.title_alternative.equalsIgnoreCase(movie.title_franchise
_br)
                                    ||
movie.title_alternative.equalsIgnoreCase(movie.title_franchise
_alternative))) {
                     return true;

                } else {
                     return false;
                }
          }
          return false;
     }

     private void addMoviesFranchise(Movie movie, Long
idPageFacebook, List<Movie> moviesFranchise, int
likedOrWatched) {
          // Adiciona e relaciona ao usuário todos os filmes
pertencentes a franquia

          // Título da página do filme é igual ao título da
franquia
          if (movie == null && moviesFranchise != null) {
                for (Movie m : moviesFranchise) {
                     // Se o filme já foi salvo como curtido,
não será salvo novamente como curtido ou assistido
                     if
(!UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,
m.id)) {
                          if (likedOrWatched == 1) {
                               // Salva filme curtido e
relaciona-o ao usuário

     user.liked_rated_movies.add(UserLikedRatedMovieCRUD.saveU
serLikedRatedMovie(
                                         new
UserLikedRatedMovie(user, m, idPageFacebook,
Float.valueOf(5))));
                          } else {
                               // Se o filme já foi salvo como
assistido, não será salvo novamente como curtido ou assistido
```

```
                                if
(!UserWatchedMovieCRUD.hasMovie(user.id_user_facebook, m.id))
{
                                     // Salva filme assistido e
relaciona-o ao usuário

     user.watched_movies.add(UserWatchedMovieCRUD

     .saveUserWatchedMovie(new UserWatchedMovie(user, m,
idPageFacebook)));
                                }
                           }
                      }
                }
                // Título do primeiro filme da franquia é igual
ao título da franquia
           } else {
                List<Movie> moviesByTitleFranchiseOriginal =
MovieCRUD

     .getMoviesByTitleFranchiseOriginal(movie.title_franchise_
original);
                for (Movie m : moviesByTitleFranchiseOriginal)
{
                     // Se o filme já foi salvo como curtido,
não será salvo novamente como curtido ou assistido
                     if
(!UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,
m.id)) {
                          if (likedOrWatched == 1) {
                               // Salva filme curtido e
relaciona-o ao usuário

     user.liked_rated_movies.add(UserLikedRatedMovieCRUD.saveU
serLikedRatedMovie(
                                          new
UserLikedRatedMovie(user, m, idPageFacebook,
Float.valueOf(5))));
                          } else {
                               // Se o filme já foi salvo como
assistido, não será salvo novamente como curtido ou assistido
                               if
(!UserWatchedMovieCRUD.hasMovie(user.id_user_facebook, m.id))
{
                                     // Salva filme assistido e
relaciona-o ao usuário

     user.watched_movies.add(UserWatchedMovieCRUD

     .saveUserWatchedMovie(new UserWatchedMovie(user, m,
idPageFacebook)));
```

```
                                    }
                               }
                          }
                     }
                }
           }

}
```

MovieMe/app/controllers/RecommenderController.java

```
package controllers;

import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;
import org.pac4j.core.profile.CommonProfile;
import org.pac4j.oauth.profile.facebook.FacebookProfile;
import org.pac4j.play.java.JavaController;

import crud.EvaluationCRUD;
import crud.MovieCRUD;
import crud.RecommendationCRUD;
import crud.RecommendedMovieCRUD;
import crud.UserCRUD;
import models.Recommendation;
import models.RecommendedMovie;
import models.User;
import play.Routes;
import play.mvc.Result;
import recommender.MovieRecommender;
import views.html.*;

public class RecommenderController extends JavaController {

     public static Result recommender() throws TasteException,
IOException {
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
          final String urlFacebook =
getRedirectAction("FacebookClient", "/?0").getLocation();
          // Verifica se o usuário está logado
          if (facebookProfile == null) {
               return ok(login.render(urlFacebook));
          } else {
               // Identifica o usuário
```

```
                User user =
UserCRUD.getUserByFacebookId(Long.parseLong(facebookProfile.ge
tId()));
                // Retorna a recomendação feita anteriormente
ao usuário, se houver
                Recommendation currentRecommendation =
RecommendationCRUD

     .getRecommendationByUserFacebookId(user.id_user_facebook)
;

                // Verifica se há dados suficiente para
realizas as recomendações
                if (user.liked_rated_movies.size() != 0 ||
user.watched_movies.size() != 0) {
                     // Verifica se já foi realizada uma
recomendação ao usuário
                     if (currentRecommendation == null) {
                          // Realiza uma recomendação de filmes
ao usuário
                          MovieRecommender movieRecommender =
new MovieRecommender(user);
                          currentRecommendation =
RecommendationCRUD.saveRecommendation(new
Recommendation(user));
                          List<RecommendedItem>
recommendedItems = movieRecommender.getRecommendations();
                          for (RecommendedItem recommendedItem
: recommendedItems) {

     currentRecommendation.movies.add(RecommendedMovieCRUD.sav
eRecommendedMovie(new RecommendedMovie(
                                             currentRecommendation,
MovieCRUD.getMovieById(recommendedItem.getItemID()),

     recommendedItem.getValue(), -1)));
                          }
                          return
ok(recommendations.render(true, true,
currentRecommendation.movies, user.first_name,
                                        (user.total_liked_movies !=
null ? user.total_liked_movies : 0),
                                        (user.total_watched_movies
!= null ? user.total_watched_movies : 0),
                                        (user.total_rated_movies !=
null ? user.total_rated_movies : 0),

     (user.total_liked_actors_directors != null ?
user.total_liked_actors_directors : 0)));
                     } else {
```

```
                         // Já foi realizada uma recomendação
ao usuário, sendo assim ela é apenas retornada
                         return
ok(recommendations.render(true, true,
currentRecommendation.movies, user.first_name,
                                   (user.total_liked_movies !=
null ? user.total_liked_movies : 0),
                                   (user.total_watched_movies
!= null ? user.total_watched_movies : 0),
                                   (user.total_rated_movies !=
null ? user.total_rated_movies : 0),

     (user.total_liked_actors_directors != null ?
user.total_liked_actors_directors : 0)));
                    }
               } else {
                    // Retorna mensagem de erro de dados
insuficientes para realizar recomendações
                    return ok(recommendations.render(false,
false, null, user.first_name, 0, 0, 0, 0));
               }
          }
     }

     public static Result movieDetails(long id_movie) {
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
          final String urlFacebook =
getRedirectAction("FacebookClient", "/?0").getLocation();
          // Verifica se o usuário está logado
          if (facebookProfile == null) {
               return ok(login.render(urlFacebook));
          } else {
               // Retorna página com detalhes do filme
recomendado
               return
ok(movie.render(MovieCRUD.getMovieById(id_movie),

     UserCRUD.getUserByFacebookId(Long.parseLong(facebookProfi
le.getId())).first_name));
          }
     }

     public static Result saveRecommendedMovieApproved(long
id_movie, int approved) {
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
```

```
          final String urlFacebook =
getRedirectAction("FacebookClient", "/?0").getLocation();
          // Verifica se o usuário está logado
          if (facebookProfile == null) {
               return ok(login.render(urlFacebook));
          } else {
               // Salva aprovação ou reprovação do usuário ao
filme recomendado
               // (0) - filme recomendado reprovado
               // (1) - filme recomendado aprovado
               RecommendedMovie recommendedMovie =
RecommendedMovieCRUD.getRecommendedMovieByRecommendationIdAndM
ovieId(

     RecommendationCRUD.getRecommendationByUserFacebookId(

     UserCRUD.getUserByFacebookId(Long.parseLong(facebookProfi
le.getId())).id_user_facebook).id,
                         id_movie);
               recommendedMovie.approved = approved;

     RecommendedMovieCRUD.updateRecommendedMovieApproved(recom
mendedMovie);
               return ok();
          }
     }

     public static Result
saveUserRecommendationEvaluationOpinion(Long id_evaluation,
String evaluation_opinion) {
          final CommonProfile commonProfile =
getUserProfile();
          final FacebookProfile facebookProfile =
(FacebookProfile) commonProfile;
          final String urlFacebook =
getRedirectAction("FacebookClient", "/?0").getLocation();
          // Verifica se o usuário está logado
          if (facebookProfile == null) {
               return ok(login.render(urlFacebook));
          } else {
               // Salva avaliação feita pelo usuário em
relação a todas as recomendações de filmes
               // (1) - Excelente / (2) - Bom / (3) - Regular
/ (4) - Ruim
               // Comentário é opcional
               Recommendation currentRecommendation =
RecommendationCRUD.getRecommendationByUserFacebookId(

     UserCRUD.getUserByFacebookId(Long.parseLong(facebookProfi
le.getId())).id_user_facebook);
               // Avaliação
```

```
                currentRecommendation.evaluation =
EvaluationCRUD.getEvaluationById(id_evaluation);
                if (evaluation_opinion != null &&
evaluation_opinion != "") {
                     // Comentário
                     currentRecommendation.evaluation_opinion =
evaluation_opinion;
                }

     RecommendationCRUD.updateRecommendationEvaluationOpinion(
currentRecommendation);
                return ok();
           }
     }

     public static Result javascriptRoutes() {
           // Necessário para realizar as operações em ajax
           response().setContentType("text/javascript");
           return ok(Routes.javascriptRouter("jsRoutes",

     routes.javascript.RecommenderController.saveUserRecommend
ationEvaluationOpinion()));
     }
}
```

package crud

MovieMe/app/crud/ActorCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Actor;
import models.Movie;
import models.MovieActor;
import models.User;
import models.UserLikedActor;

public class ActorCRUD {

     public static List<Actor> listAllActors() {
           return Actor.find.where().orderBy("id").findList();
     }

     public static void saveActor(Actor actor) {
           actor.save();
     }

     public static Actor getActorById(long id) {
```

```
          return Actor.find.byId(id);
     }

     public static List<Actor> getActorsByName(String name) {
          return Actor.find.where().like("name",
name).findList();
     }

     public static Actor getActorByIdImdb(String id_imdb) {
          return Actor.find.where().like("id_imdb",
id_imdb).findUnique();
     }

     public static List<Movie> getMoviesByActor(Actor actor) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieActor movieActor : actor.movies) {
               movies.add(movieActor.movie);
          }
          return movies;
     }

     public static List<User> getUsersByActor(Actor actor) {
          List<User> users = new ArrayList<User>();
          for (UserLikedActor userLikedActor : actor.users) {
               users.add(userLikedActor.user);
          }
          return users;
     }

}
```

MovieMe/app/crud/CountryCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Country;
import models.Movie;
import models.MovieCountry;
import models.User;
import models.UserMovieCountry;

public class CountryCRUD {

     public static List<Country> listAllCountries() {
          return
Country.find.where().orderBy("id").findList();
     }
```

```
     public static void saveCountry(Country country) {
          country.save();
     }

     public static Country getCountryById(long id) {
          return Country.find.byId(id);
     }

     public static Country getCountryByNameEn(String name_en)
{
          return Country.find.where().like("name_en",
name_en).findUnique();
     }

     public static Country getCountryByNameBr(String name_br)
{
          return Country.find.where().like("name_br",
name_br).findUnique();
     }

     public static List<Movie> getMoviesByCountry(Country
country) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieCountry movieCountry : country.movies) {
               movies.add(movieCountry.movie);
          }
          return movies;
     }

     public static List<User> getUsersByCountry(Country
country) {
          List<User> users = new ArrayList<User>();
          for (UserMovieCountry userMovieCountry :
country.users) {
               users.add(userMovieCountry.user);
          }
          return users;
     }

}
```

MovieMe/app/crud/DirectorCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Director;
```

```
import models.Movie;
import models.MovieDirector;
import models.User;
import models.UserLikedDirector;

public class DirectorCRUD {

     public static List<Director> listAllDirectors() {
          return
Director.find.where().orderBy("id").findList();
     }

     public static void saveDirector(Director director) {
          director.save();
     }

     public static Director getDirectorById(long id) {
          return Director.find.byId(id);
     }

     public static List<Director> getDirectorsByName(String
name) {
          return Director.find.where().like("name",
name).findList();
     }

     public static Director getDirectorByIdImdb(String
id_imdb) {
          return Director.find.where().like("id_imdb",
id_imdb).findUnique();
     }

     public static List<Movie> getMoviesByDirector(Director
director) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieDirector movieDirector : director.movies)
{
               movies.add(movieDirector.movie);
          }
          return movies;
     }

     public static List<User> getUsersByDirector(Director
director) {
          List<User> a = new ArrayList<User>();
          for (UserLikedDirector userLikedDirector :
director.users) {
               a.add(userLikedDirector.user);
          }
          return a;
     }
```

```
}
```

MovieMe/app/crud/EvaluationCRUD.java

```
package crud;

import java.util.List;

import models.Evaluation;

public class EvaluationCRUD {

     public static List<Evaluation> listAllEvaluations() {
          return
Evaluation.find.where().orderBy("id").findList();
     }

     public static void saveEvaluation(Evaluation evaluation)
{
          evaluation.save();
     }

     public static Evaluation getEvaluationById(long id) {
          return Evaluation.find.byId(id);
     }

     public static Evaluation getEvaluationByNameEn(String
name_en) {
          return Evaluation.find.where().like("name_en",
name_en).findUnique();
     }

     public static Evaluation getEvaluationByNameBr(String
name_br) {
          return Evaluation.find.where().like("name_br",
name_br).findUnique();
     }

}
```

MovieMe/app/crud/GenreCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Genre;
import models.Movie;
import models.MovieGenre;
```

```
import models.User;
import models.UserMovieGenre;

public class GenreCRUD {

     public static List<Genre> listAllGenres() {
          return Genre.find.where().orderBy("id").findList();
     }

     public static void saveGenre(Genre genre) {
          genre.save();
     }

     public static Genre getGenreById(long id) {
          return Genre.find.byId(id);
     }

     public static Genre getGenreByNameEn(String name_en) {
          return Genre.find.where().like("name_en",
name_en).findUnique();
     }

     public static Genre getGenreByNameBr(String name_br) {
          return Genre.find.where().like("name_br",
name_br).findUnique();
     }

     public static List<Movie> getMoviesByGenre(Genre genre) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieGenre movieGenre : genre.movies) {
               movies.add(movieGenre.movie);
          }
          return movies;
     }

     public static List<User> getUsersByGenre(Genre genre) {
          List<User> users = new ArrayList<User>();
          for (UserMovieGenre userMovieGenre : genre.users) {
               users.add(userMovieGenre.user);
          }
          return users;
     }

}
```

MovieMe/app/crud/MovieActorCRUD.java

```
package crud;

import java.util.ArrayList;
```

```
import java.util.List;

import models.Actor;
import models.Movie;
import models.MovieActor;

public class MovieActorCRUD {

     public static List<MovieActor> listAllMoviesActors() {
          return
MovieActor.find.where().orderBy("id").findList();
     }

     public static void saveMovieActor(MovieActor movieActor)
{
          movieActor.save();
     }

     public static MovieActor getMovieActorById(long id) {
          return MovieActor.find.byId(id);
     }

     public static List<MovieActor>
getMoviesActorsByMovieId(long id_movie) {
          return MovieActor.find.where().eq("id_movie",
id_movie).findList();
     }

     public static List<MovieActor>
getMoviesActorsByActorId(long id_actor) {
          return MovieActor.find.where().eq("id_actor",
id_actor).findList();
     }

     public static List<Actor> getActorsByMovieId(long
id_movie) {
          List<Actor> actors = new ArrayList<Actor>();
          for (MovieActor movieActor :
getMoviesActorsByMovieId(id_movie)) {
               actors.add(movieActor.actor);
          }
          return actors;
     }

     public static List<Movie> getMoviesByActorId(long
id_actor) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieActor movieActor :
getMoviesActorsByActorId(id_actor)) {
               movies.add(movieActor.movie);
          }
```

```
          return movies;
     }

}
```

MovieMe/app/crud/MovieCountryCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Country;
import models.Movie;
import models.MovieCountry;

public class MovieCountryCRUD {

     public static List<MovieCountry> listAllMoviesCountries()
{
          return
MovieCountry.find.where().orderBy("id").findList();
     }

     public static void saveMovieCountry(MovieCountry
movieCountry) {
          movieCountry.save();
     }

     public static MovieCountry getMovieCountryById(long id) {
          return MovieCountry.find.byId(id);
     }

     public static List<MovieCountry>
getMoviesCountriesByMovieId(long id_movie) {
          return MovieCountry.find.where().eq("id_movie",
id_movie).findList();
     }

     public static List<MovieCountry>
getMoviesCountriesByCountryId(long id_country) {
          return MovieCountry.find.where().eq("id_country",
id_country).findList();
     }

     public static List<Country> getCountriesByMovieId(long
id_movie) {
          List<Country> countries = new ArrayList<Country>();
          for (MovieCountry movieCountry :
getMoviesCountriesByMovieId(id_movie)) {
```

```
                countries.add(movieCountry.country);
            }
            return countries;
        }

        public static List<Movie> getMoviesByCountryId(long
id_country) {
            List<Movie> movies = new ArrayList<Movie>();
            for (MovieCountry movieCountry :
getMoviesCountriesByCountryId(id_country)) {
                movies.add(movieCountry.movie);
            }
            return movies;
        }
}
```

MovieMe/app/crud/MovieCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import com.avaje.ebean.Expr;

import models.Actor;
import models.Country;
import models.Director;
import models.Genre;
import models.Movie;
import models.MovieActor;
import models.MovieCountry;
import models.MovieDirector;
import models.MovieGenre;
import models.MovieSimilar;
import models.RecommendedMovie;
import models.User;
import models.UserLikedRatedMovie;
import models.UserWatchedMovie;

public class MovieCRUD {

    public static List<Movie> listAllMovies() {
        return Movie.find.where().orderBy("id").findList();
    }

    public static int getTotalOfMovies() {
        return Movie.find.findRowCount();
    }
```

```
     public static void saveMovie(Movie movie) {
          movie.save();
     }

     public static Movie getMovieById(long id) {
          return Movie.find.byId(id);
     }

     public static List<Movie> getMoviesByTitleOriginal(String
title_original) {
          return Movie.find.where().like("title_original",
title_original).findList();
     }

     public static Movie getMovieByTitleOriginalAndYear(String
title_original, int year) {
          return
Movie.find.where().and(Expr.like("title_original",
title_original), Expr.eq("year", year)).findUnique();
     }

     public static List<Movie> getMoviesByTitleBr(String
title_br) {
          return Movie.find.where().like("title_br",
title_br).findList();
     }

     public static Movie getMovieByTitleBrAndYear(String
title_br, int year) {
          return Movie.find.where().and(Expr.like("title_br",
title_br), Expr.eq("year", year)).findUnique();
     }

     public static List<Movie> getMoviesByTitleEn(String
title_en) {
          return Movie.find.where().like("title_en",
title_en).findList();
     }

     public static Movie getMovieByTitleEnAndYear(String
title_en, int year) {
          return Movie.find.where().and(Expr.like("title_en",
title_en), Expr.eq("year", year)).findUnique();
     }

     public static List<Movie>
getMoviesByTitleAlternative(String title_alternative) {
          return Movie.find.where().like("title_alternative",
title_alternative).findList();
     }
```

```
     public static Movie
getMovieByTitleAlternativeAndYear(String title_alternative,
int year) {
          return
Movie.find.where().and(Expr.like("title_alternative",
title_alternative), Expr.eq("year", year))
                    .findUnique();
     }

     public static List<Movie>
getMoviesByTitleFranchiseOriginal(String
title_franchise_original) {
          return
Movie.find.where().like("title_franchise_original",
title_franchise_original).findList();
     }

     public static List<Movie>
getMoviesByTitleFranchiseBr(String title_franchise_br) {
          return Movie.find.where().like("title_franchise_br",
title_franchise_br).findList();
     }

     public static List<Movie>
getMoviesByTitleFranchiseAlternative(String
title_franchise_alternative) {
          return
Movie.find.where().like("title_franchise_alternative",
title_franchise_alternative).findList();
     }

     public static List<Movie>
getMoviesByIsMovieFranchise(Boolean is_movie_franchise) {
          return Movie.find.where().eq("is_movie_franchise",
1).findList();
     }

     public static List<Movie>
getMoviesByIsNotMovieFranchise(Boolean is_movie_franchise) {
          return Movie.find.where().eq("is_movie_franchise",
0).findList();
     }

     public static List<Movie> getMoviesByYear(int year) {
          return Movie.find.where().eq("year",
year).findList();
     }

     public static Movie getMovieByIdImdb(String idImdb) {
          return Movie.find.where().like("id_imdb",
idImdb).findUnique();
```

```
     }

     public static Movie
getPreviousFranchiseMovieByMovieId(long id) {
          Movie movie = MovieCRUD.getMovieById(id);
          return
Movie.find.where().and(Expr.like("title_franchise_original",
movie.title_franchise_original),
                    Expr.eq("order_franchise",
(movie.order_franchise - 1))).findUnique();
     }

     public static List<Movie>
getNextFranchiseMoviesByMovieId(long id) {
          Movie movie = MovieCRUD.getMovieById(id);
          return
Movie.find.where().and(Expr.like("title_franchise_original",
movie.title_franchise_original),
                    Expr.gt("order_franchise",
movie.order_franchise)).findList();
     }

     public static List<Actor> getActorsByMovie(Movie movie) {
          List<Actor> actors = new ArrayList<Actor>();
          for (MovieActor movieActor : movie.actors) {
               actors.add(movieActor.actor);
          }
          return actors;
     }

     public static List<Actor> getActorsByMovieId(long id) {
          List<Actor> actors = new ArrayList<Actor>();
          for (MovieActor movieActor :
getMovieById(id).actors) {
               actors.add(movieActor.actor);
          }
          return actors;
     }

     public static List<Country> getCountriesByMovie(Movie
movie) {
          List<Country> countries = new ArrayList<Country>();
          for (MovieCountry movieCountry : movie.countries) {
               countries.add(movieCountry.country);
          }
          return countries;
     }

     public static List<Country> getCountriesByMovieId(long
id) {
          List<Country> countries = new ArrayList<Country>();
```

```
          for (MovieCountry movieCountry :
getMovieById(id).countries) {
               countries.add(movieCountry.country);
          }
          return countries;
     }

     public static List<Director> getDirectorsByMovie(Movie
movie) {
          List<Director> directors = new
ArrayList<Director>();
          for (MovieDirector movieDirector : movie.directors)
{
               directors.add(movieDirector.director);
          }
          return directors;
     }

     public static List<Director> getDirectorsByMovieId(long
id) {
          List<Director> directors = new
ArrayList<Director>();
          for (MovieDirector movieDirector :
getMovieById(id).directors) {
               directors.add(movieDirector.director);
          }
          return directors;
     }

     public static List<Genre> getGenresByMovie(Movie movie) {
          List<Genre> genres = new ArrayList<Genre>();
          for (MovieGenre movieGenre : movie.genres) {
               genres.add(movieGenre.genre);
          }
          return genres;
     }

     public static List<Genre> getGenresByMovieId(long id) {
          List<Genre> genres = new ArrayList<Genre>();
          for (MovieGenre movieGenre :
getMovieById(id).genres) {
               genres.add(movieGenre.genre);
          }
          return genres;
     }

     public static List<Movie> getSimilarMoviesByMovie(Movie
movie) {
          List<Movie> moviesSimilar = new ArrayList<Movie>();
          for (MovieSimilar movieSimilar :
movie.movies_similar) {
```

```
                moviesSimilar.add(movieSimilar.movie_similar);
            }
            return moviesSimilar;
        }

        public static List<Movie> getSimilarMoviesByMovieId(long
id) {
            List<Movie> moviesSimilar = new ArrayList<Movie>();
            for (MovieSimilar movieSimilar :
getMovieById(id).movies_similar) {
                moviesSimilar.add(movieSimilar.movie_similar);
            }
            return moviesSimilar;
        }

        public static List<Movie>
getRecommendedMoviesByMovie(Movie movie) {
            List<Movie> movies = new ArrayList<Movie>();
            for (RecommendedMovie recommendedMovie :
movie.recommendations) {
                movies.add(recommendedMovie.movie);
            }
            return movies;
        }

        public static List<Movie>
getRecommendedMoviesByMovieId(long id) {
            List<Movie> movies = new ArrayList<Movie>();
            for (RecommendedMovie recommendedMovie :
getMovieById(id).recommendations) {
                movies.add(recommendedMovie.movie);
            }
            return movies;
        }

        public static List<User>
getUsersLikedRatedMovieByMovie(Movie movie) {
            List<User> users = new ArrayList<User>();
            for (UserLikedRatedMovie userLikedRatedMovie :
movie.users_liked_rated) {
                users.add(userLikedRatedMovie.user);
            }
            return users;
        }

        public static List<User>
getUsersLikedRatedMovieByMovieId(long id) {
            List<User> users = new ArrayList<User>();
            for (UserLikedRatedMovie userLikedRatedMovie :
getMovieById(id).users_liked_rated) {
                users.add(userLikedRatedMovie.user);
```

```
          }
          return users;
     }

     public static List<User>
getUsersWatchedMovieByMovie(Movie movie) {
          List<User> users = new ArrayList<User>();
          for (UserWatchedMovie userWatchedMovie :
movie.users_watched) {
               users.add(userWatchedMovie.user);
          }
          return users;
     }

     public static List<User>
getUsersWatchedMovieByMovieId(long id) {
          List<User> users = new ArrayList<User>();
          for (UserWatchedMovie userWatchedMovie :
getMovieById(id).users_watched) {
               users.add(userWatchedMovie.user);
          }
          return users;
     }

}
```

MovieMe/app/crud/MovieDirectorCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Director;
import models.Movie;
import models.MovieDirector;

public class MovieDirectorCRUD {

     public static List<MovieDirector>
listAllMoviesDirectors() {
          return
MovieDirector.find.where().orderBy("id").findList();
     }

     public static void saveMovieDirector(MovieDirector
movieDirector) {
          movieDirector.save();
     }
```

```
     public MovieDirector getMovieDirectorById(long id) {
          return MovieDirector.find.byId(id);
     }

     public static List<MovieDirector>
getMoviesDirectorsByMovieId(long id_movie) {
          return MovieDirector.find.where().eq("id_movie",
id_movie).findList();
     }

     public static List<MovieDirector>
getMoviesDirectorsByDirectorId(long id_director) {
          return MovieDirector.find.where().eq("id_director",
id_director).findList();
     }

     public static List<Director> getDirectorsByMovieId(long
id_movie) {
          List<Director> directors = new
ArrayList<Director>();
          for (MovieDirector movieDirector :
getMoviesDirectorsByMovieId(id_movie)) {
               directors.add(movieDirector.director);
          }
          return directors;
     }

     public static List<Movie> getMoviesByDirectorId(long
id_director) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieDirector movieDirector :
getMoviesDirectorsByDirectorId(id_director)) {
               movies.add(movieDirector.movie);
          }
          return movies;
     }

}
```

MovieMe/app/crud/MovieGenreCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Genre;
import models.Movie;
import models.MovieGenre;
```

```
public class MovieGenreCRUD {

     public static List<MovieGenre> listAllMoviesGenres() {
          return
MovieGenre.find.where().orderBy("id").findList();
     }

     public static void saveMovieGenre(MovieGenre movieGenre)
{
          movieGenre.save();
     }

     public static MovieGenre getMovieGenreById(long id) {
          return MovieGenre.find.byId(id);
     }

     public static List<MovieGenre>
getMoviesGenresByMovieId(long id_movie) {
          return MovieGenre.find.where().eq("id_movie",
id_movie).findList();
     }

     public static List<MovieGenre>
getMoviesGenresByGenreId(long id_genre) {
          return MovieGenre.find.where().eq("id_genre",
id_genre).findList();
     }

     public static List<Genre> getGenresByMovieId(long
id_movie) {
          List<Genre> genres = new ArrayList<Genre>();
          for (MovieGenre movieGenre :
getMoviesGenresByMovieId(id_movie)) {
               genres.add(movieGenre.genre);
          }
          return genres;
     }

     public static List<Movie> getMoviesByGenreId(long
id_genre) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieGenre movieGenre :
getMoviesGenresByGenreId(id_genre)) {
               movies.add(movieGenre.movie);
          }
          return movies;
     }

}
```

MovieMe/app/crud/MovielensLinkCRUD.java

```
package crud;

import java.util.List;

import models.MovielensLink;

public class MovielensLinkCRUD {

     public static List<MovielensLink> listAllMovielensLinks()
{
          return
MovielensLink.find.where().orderBy("id").findList();
     }

     public static List<MovielensLink>
listAllMovielensLinksWithRange(int first_row, int max_rows) {
          return
MovielensLink.find.where().setFirstRow(first_row).setMaxRows(m
ax_rows).findList();
     }

     public static void saveMovielensLink(MovielensLink
movielensLink) {
          movielensLink.save();
     }

     public static MovielensLink
getMovielensLinkByMovieId(long id_movie) {
          return MovielensLink.find.byId(id_movie);
     }

     public static MovielensLink
getMovielensLinkByIdImdb(String id_imdb) {
          return MovielensLink.find.where().like("id_imdb",
id_imdb).findUnique();
     }

     public static MovielensLink
getMovielensLinkByIdTmdb(String id_tmdb) {
          return MovielensLink.find.where().like("id_tmdb",
id_tmdb).findUnique();
     }

}
```

MovieMe/app/crud/MovielensMovieCRUD.java

```
package crud;
```

```
import java.util.List;

import models.MovielensMovie;

public class MovielensMovieCRUD {

     public static List<MovielensMovie>
listAllMovielensMovies() {
          return
MovielensMovie.find.where().orderBy("id").findList();
     }

     public static void saveMovielensMovie(MovielensMovie
movielensMovie) {
          movielensMovie.save();
     }

     public static MovielensMovie getMovielensMovieById(long
id) {
          return MovielensMovie.find.byId(id);
     }

     public static MovielensMovie
getMovielensMovieByTitle(String title) {
          return MovielensMovie.find.where().like("title",
title).findUnique();
     }

}
```

MovieMe/app/crud/MovielensRatingCRUD.java

```
package crud;

import java.util.List;

import com.avaje.ebean.Expr;

import models.MovielensRating;

public class MovielensRatingCRUD {

     public static List<MovielensRating>
listAllMovielensRatings() {
          return
MovielensRating.find.where().orderBy("id").findList();
     }
```

```
     public static void saveMovielensRating(MovielensRating
movielensRating) {
          movielensRating.save();
     }

     public static MovielensRating
getMovielensRatingByUserIdAndMovieId(long id_user, long
id_movie) {
          return
MovielensRating.find.where().and(Expr.eq("id_user", id_user),
Expr.eq("id_movie", id_movie))
                    .findUnique();
     }

     public static List<MovielensRating>
getMovielensRatingsByUserId(long id_user) {
          return MovielensRating.find.where().eq("id_user",
id_user).findList();
     }

}
```

MovieMe/app/crud/MovieSimilarCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Movie;
import models.MovieSimilar;

public class MovieSimilarCRUD {

     public static List<MovieSimilar> listAllMoviesSimilar() {
          return
MovieSimilar.find.where().orderBy("id").findList();
     }

     public static void saveMovieSimilar(MovieSimilar
movieSimilar) {
          movieSimilar.save();
     }

     public static MovieSimilar getMovieSimilarById(long id) {
          return MovieSimilar.find.byId(id);
     }

     public static List<MovieSimilar>
getMoviesSimilarByMovieId(long id_movie) {
```

```
          return MovieSimilar.find.where().eq("id_movie",
id_movie).findList();
     }

     public static List<MovieSimilar>
getMoviesSimilarWithRangeByMovieId(long id_movie, int
max_rows) {
          return MovieSimilar.find.where().eq("id_movie",
id_movie).setMaxRows(max_rows).findList();
     }

     public static List<Movie> getSimilarMoviesByMovieId(long
id_movie) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieSimilar movieSimilar :
getMoviesSimilarByMovieId(id_movie)) {
               movies.add(movieSimilar.movie_similar);
          }
          return movies;
     }

     public static List<Movie>
getSimilarMoviesWithRangeByMovieId(long id_movie, int
max_rows) {
          List<Movie> movies = new ArrayList<Movie>();
          for (MovieSimilar movieSimilar :
getMoviesSimilarWithRangeByMovieId(id_movie, max_rows)) {
               movies.add(movieSimilar.movie_similar);
          }
          return movies;
     }

}
```

MovieMe/app/crud/RecommendationCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Movie;
import models.Recommendation;
import models.RecommendedMovie;

public class RecommendationCRUD {

     public static List<Recommendation>
listAllRecommendations() {
```

```
          return
Recommendation.find.where().orderBy("id").findList();
     }

     public static Recommendation
saveRecommendation(Recommendation recommendation) {
          recommendation.save();
          return recommendation;
     }

     public static void
updateRecommendationEvaluationOpinion(Recommendation
recommendation) {
          recommendation.update();
     }

     public static Recommendation getRecommendationById(long
id) {
          return Recommendation.find.byId(id);
     }

     public static Recommendation
getRecommendationByUserFacebookId(long id_user_facebook) {
          return
Recommendation.find.where().eq("id_user_facebook",
id_user_facebook).findUnique();
     }

     public static List<Movie>
getMoviesByRecommendation(Recommendation recommendation) {
          List<Movie> movies = new ArrayList<Movie>();
          for (RecommendedMovie recommendedMovie :
recommendation.movies) {
               movies.add(recommendedMovie.movie);
          }
          return movies;
     }

}
```

MovieMe/app/crud/RecommendedMovieCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import com.avaje.ebean.Expr;

import models.Movie;
```

```
import models.RecommendedMovie;

public class RecommendedMovieCRUD {

     public static List<RecommendedMovie>
listAllRecommendedMovies() {
          return
RecommendedMovie.find.where().orderBy("id").findList();
     }

     public static RecommendedMovie
saveRecommendedMovie(RecommendedMovie recommendedMovie) {
          recommendedMovie.save();
          return recommendedMovie;
     }

     public static void
updateRecommendedMovieApproved(RecommendedMovie
recommendedMovie) {
          recommendedMovie.update();
     }

     public static RecommendedMovie
getRecommendedMovieById(long id) {
          return RecommendedMovie.find.byId(id);
     }

     public static List<RecommendedMovie>
getRecommendedMoviesByRecommendationId(long id_recommendation)
{
          return
RecommendedMovie.find.where().eq("id_recommendation",
id_recommendation).orderBy("score DESC")
                    .findList();
     }

     public static List<RecommendedMovie>
getRecommendedMoviesByMovieId(long id_movie) {
          return RecommendedMovie.find.where().eq("id_movie",
id_movie).findList();
     }

     public static RecommendedMovie
getRecommendedMovieByRecommendationIdAndMovieId(long
id_recommendation,
               long id_movie) {
          return RecommendedMovie.find.where()
                    .and(Expr.eq("id_recommendation",
id_recommendation), Expr.eq("id_movie",
id_movie)).findUnique();
     }
```

```
     public static List<Movie>
getMoviesByRecommendationId(long id_recommendation) {
          List<Movie> movies = new ArrayList<Movie>();
          for (RecommendedMovie recommendedMovie :
getRecommendedMoviesByRecommendationId(id_recommendation)) {
               movies.add(recommendedMovie.movie);
          }
          return movies;
     }
}
```

MovieMe/app/crud/RoleCRUD.java

```
package crud;

import java.util.List;

import models.Role;

public class RoleCRUD {

     public static List<Role> listAllRoles() {
          return Role.find.where().orderBy("id").findList();
     }

     public static void saveRole(Role role) {
          role.save();
     }

     public static Role getRoleById(long id) {
          return Role.find.byId(id);
     }

     public static Role getRoleByNameEn(String name_en) {
          return Role.find.where().like("name_en",
name_en).findUnique();
     }

     public static Role getRoleByNameBr(String name_br) {
          return Role.find.where().like("name_br",
name_br).findUnique();
     }

}
```

MovieMe/app/crud/UserCRUD.java

```
package crud;
```

```
import java.util.ArrayList;
import java.util.List;

import models.Actor;
import models.Country;
import models.Director;
import models.Genre;
import models.Movie;
import models.User;
import models.UserLikedActor;
import models.UserLike;
import models.UserLikedDirector;
import models.UserLikedRatedMovie;
import models.UserMovieCountry;
import models.UserMovieGenre;
import models.UserWatchedMovie;

public class UserCRUD {

     public static List<User> listAllUsers() {
          return
User.find.where().orderBy("id_user_facebook").findList();
     }

     public static User saveUser(User user) {
          user.save();
          return getUserByFacebookId(user.id_user_facebook);
     }

     public static void updateUserTotalOfLikedMovies(User
user) {
          user.update();
     }

     public static void updateUserTotalOfWatchedMovies(User
user) {
          user.update();
     }

     public static void updateUserTotalOfRatedMovies(User
user) {
          user.update();
     }

     public static void
updateUserTotalOfLikedActorsDirectors(User user) {
          user.update();
     }

     public static User getUserByFacebookId(long
id_user_facebook) {
```

```
          return User.find.byId(id_user_facebook);
     }

     public static List<Actor> getLikedActorsByUser(User user)
{
          List<Actor> actors = new ArrayList<Actor>();
          for (UserLikedActor userLikedActor :
user.liked_actors) {
               actors.add(userLikedActor.actor);
          }
          return actors;
     }

     public static List<Director> getLikedDirectorsByUser(User
user) {
          List<Director> directors = new
ArrayList<Director>();
          for (UserLikedDirector userLikedDirector :
user.liked_directors) {
               directors.add(userLikedDirector.director);
          }
          return directors;
     }

     public static List<Movie> getLikedRatedMoviesByUser(User
user) {
          List<Movie> movies = new ArrayList<Movie>();
          for (UserLikedRatedMovie userLikedRatedMovie :
user.liked_rated_movies) {
               movies.add(userLikedRatedMovie.movie);
          }
          return movies;
     }

     public static boolean movieWasLikedOrRatedByUser(User
user, Movie movie) {
          for (UserLikedRatedMovie userLikedRatedMovie :
user.liked_rated_movies) {
               if (userLikedRatedMovie.movie.equals(movie)) {
                    return true;
               }
          }
          return false;
     }

     public static List<Movie> getWatchedMoviesByUser(User
user) {
          List<Movie> movies = new ArrayList<Movie>();
          for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {
               movies.add(userWatchedMovie.movie);
```

```
          }
          return movies;
     }

     public static boolean movieWasWatchedByUser(User user,
Movie movie) {
          for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {
               if (userWatchedMovie.movie.equals(movie)) {
                    return true;
               }
          }
          return false;
     }

     public static List<Country> getCountriesByUser(User user)
{
          List<Country> countries = new ArrayList<Country>();
          for (UserMovieCountry userMovieCountry :
user.movies_countries) {
               countries.add(userMovieCountry.country);
          }
          return countries;
     }

     public static List<Country>
getCountriesWithRangeByUser(User user, int limit) {
          List<Country> countries = new ArrayList<Country>();
          for (int i = 0; i <= limit; i++) {

     countries.add(user.movies_countries.get(i).country);
          }
          return countries;
     }

     public static List<Genre> getGenresByUser(User user) {
          List<Genre> genres = new ArrayList<Genre>();
          for (UserMovieGenre userMovieGenre :
user.movies_genres) {
               genres.add(userMovieGenre.genre);
          }
          return genres;
     }

     public static List<Genre> getGenresWithRangeByUser(User
user, int limit) {
          List<Genre> genres = new ArrayList<Genre>();
          for (int i = 0; i <= limit; i++) {
               genres.add(user.movies_genres.get(i).genre);
          }
          return genres;
```

```
     }

     public static List<Long>
getLikedPageFacebookIdsByUser(User user) {
          List<Long> idsFacebookPages = new ArrayList<Long>();
          for (UserLike userLike : user.likes) {

     idsFacebookPages.add(userLike.id_page_facebook);
          }
          return idsFacebookPages;
     }

     public static List<UserLike>
getUsersLikesByUserAndCategory(User user, String
page_category) {
          List<UserLike> userLikesCategory = new
ArrayList<UserLike>();
          for (UserLike userLike : user.likes) {
               if
(userLike.category_page_facebook.equals(page_category)) {
                    userLikesCategory.add(userLike);
               }
          }
          return userLikesCategory;
     }

}
```

MovieMe/app/crud/UserLikedActorCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Actor;
import models.UserLikedActor;

public class UserLikedActorCRUD {

     public static List<UserLikedActor>
listAllUsersLikedActors() {
          return
UserLikedActor.find.where().orderBy("id").findList();
     }

     public static UserLikedActor
saveUserLikedActor(UserLikedActor userLikedActor) {
          userLikedActor.save();
          return userLikedActor;
```

```
     }

     public static UserLikedActor getUserLikedActorById(long
id) {
          return UserLikedActor.find.byId(id);
     }

     public static int
getTotalOfLikedActorsByUserFacebookId(long id_user_facebook) {
          return
UserLikedActor.find.where().eq("id_user_facebook",
id_user_facebook).findRowCount();
     }

     public static List<UserLikedActor>
getUsersLikedActorsByUserFacebookId(long id_user_facebook) {
          return
UserLikedActor.find.where().eq("id_user_facebook",
id_user_facebook).findList();
     }

     public static List<Actor>
getLikedActorsByUserFacebookId(long id_user_facebook) {
          List<Actor> actors = new ArrayList<Actor>();
          for (UserLikedActor userLikedActor :
getUsersLikedActorsByUserFacebookId(id_user_facebook)) {
               actors.add(userLikedActor.actor);
          }
          return actors;
     }

     public static boolean hasActor(long id_user_facebook,
long id_actor) {
          for (UserLikedActor userLikedActor :
getUsersLikedActorsByUserFacebookId(id_user_facebook)) {
               if (userLikedActor.actor.id == id_actor) {
                    return true;
               }
          }
          return false;
     }

}
```

MovieMe/app/crud/UserLikeDAO.java

```
package crud;

import java.util.List;
```

```
import models.UserLike;

public class UserLikeDAO {

     public static List<UserLike> listAllUsersLikes() {
          return
UserLike.find.where().orderBy("id").findList();
     }

     public static UserLike saveUserLike(UserLike userLike) {
          userLike.save();
          return userLike;
     }

     public static UserLike getUserLikeById(long id) {
          return UserLike.find.byId(id);
     }

     public static List<UserLike>
getUsersLikesByUserFacebookId(long id_user_facebook) {
          return UserLike.find.where().eq("id_user_facebook",
id_user_facebook).findList();
     }

     public static boolean hasFacebookPage(long
id_user_facebook, long id_page_facebook) {
          for (UserLike userLike :
getUsersLikesByUserFacebookId(id_user_facebook)) {
               if (userLike.id_page_facebook ==
id_page_facebook) {
                    return true;
               }
          }
          return false;
     }

}
```

MovieMe/app/crud/UserLikedDirectorCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Director;
import models.UserLikedDirector;

public class UserLikedDirectorCRUD {
```

```
     public static List<UserLikedDirector>
listAllUsersLikedDirectors() {
          return
UserLikedDirector.find.where().orderBy("id").findList();
     }

     public static UserLikedDirector
saveUserLikedDirector(UserLikedDirector userLikedDirector) {
          userLikedDirector.save();
          return userLikedDirector;
     }

     public static UserLikedDirector
getUserLikedDirectorById(long id) {
          return UserLikedDirector.find.byId(id);
     }

     public static int
getTotalOfLikedDirectorsByUserFacebookId(long
id_user_facebook) {
          return
UserLikedDirector.find.where().eq("id_user_facebook",
id_user_facebook).findRowCount();
     }

     public static List<UserLikedDirector>
getUsersLikedDirectorsByUserFacebookId(long id_user_facebook)
{
          return
UserLikedDirector.find.where().eq("id_user_facebook",
id_user_facebook).findList();
     }

     public static List<Director>
getLikedDirectorsByUserFacebookId(long id_user_facebook) {
          List<Director> directors = new
ArrayList<Director>();
          for (UserLikedDirector userLikedDirector :
getUsersLikedDirectorsByUserFacebookId(id_user_facebook)) {
               directors.add(userLikedDirector.director);
          }
          return directors;
     }

     public static boolean hasDirector(long id_user_facebook,
long id_director) {
          for (UserLikedDirector userLikedDirector :
getUsersLikedDirectorsByUserFacebookId(id_user_facebook)) {
               if (userLikedDirector.director.id ==
id_director) {
                    return true;
```

```
                }
            }
            return false;
      }
}
```

MovieMe/app/crud/UserLikedRatedMovieCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import com.avaje.ebean.Expr;

import models.Movie;
import models.UserLikedRatedMovie;

public class UserLikedRatedMovieCRUD {

      public static List<UserLikedRatedMovie>
listAllUsersLikedRatedMovies() {
            return
UserLikedRatedMovie.find.where().orderBy("id").findList();
      }

      public static UserLikedRatedMovie
saveUserLikedRatedMovie(UserLikedRatedMovie
userLikedRatedMovie) {
            userLikedRatedMovie.save();
            return userLikedRatedMovie;
      }

      public static void
updateUserLikedRatedMovieRating(UserLikedRatedMovie
userLikedRatedMovie) {
            userLikedRatedMovie.update();
      }

      public static UserLikedRatedMovie
getUserLikedRatedMovieById(long id) {
            return UserLikedRatedMovie.find.byId(id);
      }

      public static int
getTotalOfLikedRatedMoviesByUserFacebookId(long
id_user_facebook) {
            return
UserLikedRatedMovie.find.where().eq("id_user_facebook",
id_user_facebook).findRowCount();
```

```
    }

    public static List<UserLikedRatedMovie>
getUsersLikedRatedMoviesByUserFacebookId(long
id_user_facebook) {
        return
UserLikedRatedMovie.find.where().eq("id_user_facebook",
id_user_facebook).findList();
    }

    public static UserLikedRatedMovie
getUserLikedRatedMovieByUserFacebookIdAndMovieId(long
id_user_facebook,
            long id_movie) {
        return UserLikedRatedMovie.find.where()
                .and(Expr.eq("id_user_facebook",
id_user_facebook), Expr.eq("id_movie",
id_movie)).findUnique();
    }

    public static List<UserLikedRatedMovie>
getUsersLikedRatedMoviesByPageFacebookId(long
id_page_facebook) {
        return
UserLikedRatedMovie.find.where().eq("id_page_facebook",
id_page_facebook).findList();
    }

    public static List<Movie>
getLikedRatedMoviesdByUserFacebookId(long id_user_facebook) {
        List<Movie> movies = new ArrayList<Movie>();
        for (UserLikedRatedMovie userLikedRatedMovie :
getUsersLikedRatedMoviesByUserFacebookId(id_user_facebook)) {
            movies.add(userLikedRatedMovie.movie);
        }
        return movies;
    }

    public static List<Movie>
getLikedRatedMoviesdByPageFacebookId(long id_page_facebook) {
        List<Movie> movies = new ArrayList<Movie>();
        for (UserLikedRatedMovie userLikedRatedMovie :
getUsersLikedRatedMoviesByPageFacebookId(id_page_facebook)) {
            movies.add(userLikedRatedMovie.movie);
        }
        return movies;
    }

    public static boolean hasMovie(long id_user, long
id_movie) {
```

```
          for (UserLikedRatedMovie userLikedRatedMovie :
getUsersLikedRatedMoviesByUserFacebookId(id_user)) {
               if (userLikedRatedMovie.movie.id == id_movie) {
                    return true;
               }
          }
          return false;
     }
}
```

MovieMe/app/crud/UserMovieCountryCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Country;
import models.UserMovieCountry;

public class UserMovieCountryCRUD {

     public static List<UserMovieCountry>
listAllUsersMoviesCountries() {
          return
UserMovieCountry.find.where().orderBy("id").findList();
     }

     public static UserMovieCountry
saveUserMovieCountry(UserMovieCountry userMovieCountry) {
          userMovieCountry.save();
          return userMovieCountry;
     }

     public static UserMovieCountry
gerUserMovieCountryById(long id) {
          return UserMovieCountry.find.byId(id);
     }

     public static UserMovieCountry
getUserMovieCountryByCountryId(long id_country) {
          return
UserMovieCountry.find.where().eq("id_country",
id_country).findUnique();
     }

     public static List<UserMovieCountry>
getTopUsersMoviesCountriesWithRangeByUserFacebookId(long
id_user_facebook,
               int max_rows) {
```

```
          return
UserMovieCountry.find.where().eq("id_user_facebook",
id_user_facebook).orderBy("quantity DESC")
                    .setMaxRows(max_rows).findList();
     }

     public static List<Country>
getTopCountriesWithRangeByUserFacebookId(long
id_user_facebook, int max_rows) {
          List<Country> countries = new ArrayList<Country>();
          for (UserMovieCountry userMovieCountry :
getTopUsersMoviesCountriesWithRangeByUserFacebookId(id_user_fa
cebook,
                    max_rows)) {
               countries.add(userMovieCountry.country);
          }
          return countries;
     }

     public static boolean hasCountry(long id_country) {
          if (getUserMovieCountryByCountryId(id_country) !=
null) {
               return true;
          } else {
               return false;
          }
     }
}
```

MovieMe/app/crud/UserMovieGenreCRUD.java

```
package crud;

import java.util.ArrayList;
import java.util.List;

import models.Genre;
import models.UserMovieGenre;

public class UserMovieGenreCRUD {

     public static List<UserMovieGenre>
listAllUsersMoviesGenres() {
          return
UserMovieGenre.find.where().orderBy("id").findList();
     }

     public static UserMovieGenre
saveUserMovieGenre(UserMovieGenre userMovieGenre) {
          userMovieGenre.save();
```

```
          return userMovieGenre;
     }

     public static UserMovieGenre getUserMovieGenreById(long
id) {
          return UserMovieGenre.find.byId(id);
     }

     public static UserMovieGenre
getUserMovieGenreByGenreId(long id_genre) {
          return UserMovieGenre.find.where().eq("id_genre",
id_genre).findUnique();
     }

     public static List<UserMovieGenre>
getTopUsersMoviesGenresWithRangeByUserFacebookId(long
id_user_facebook,
               int max_rows) {
          return
UserMovieGenre.find.where().eq("id_user_facebook",
id_user_facebook).orderBy("quantity DESC")
                    .setMaxRows(max_rows).findList();
     }

     public static List<Genre>
getTopGenresWithRangeByUserFacebookId(long id_user_facebook,
int max_rows) {
          List<Genre> genres = new ArrayList<Genre>();
          for (UserMovieGenre userMovieGenre :
getTopUsersMoviesGenresWithRangeByUserFacebookId(id_user_faceb
ook,
                    max_rows)) {
               genres.add(userMovieGenre.genre);
          }
          return genres;
     }

     public static boolean hasGenre(long id_genre) {
          if (getUserMovieGenreByGenreId(id_genre) != null) {
               return true;
          } else {
               return false;
          }
     }
}
```

MovieMe/app/crud/UserWatchedMovieCRUD.java

```
package crud;
```

```
import java.util.ArrayList;
import java.util.List;

import com.avaje.ebean.Expr;

import models.Movie;
import models.UserWatchedMovie;

public class UserWatchedMovieCRUD {

     public static List<UserWatchedMovie>
listAllUsersWatchedMovies() {
          return
UserWatchedMovie.find.where().orderBy("id").findList();
     }

     public static UserWatchedMovie
saveUserWatchedMovie(UserWatchedMovie userWatchedMovie) {
          userWatchedMovie.save();
          return userWatchedMovie;
     }

     public static UserWatchedMovie
getUserWatchedMovieById(long id) {
          return UserWatchedMovie.find.byId(id);
     }

     public static int
getTotalOfWatchedMoviesByUserFacebookId(long id_user_facebook)
{
          return
UserWatchedMovie.find.where().eq("id_user_facebook",
id_user_facebook).findRowCount();
     }

     public static List<UserWatchedMovie>
getUsersWatchedMoviesByUserFacebookId(long id_user_facebook) {
          return
UserWatchedMovie.find.where().eq("id_user_facebook",
id_user_facebook).findList();
     }

     public static UserWatchedMovie
getUserWatchedMovieByUserFacebookIdAndMovieId(long
id_user_facebook, long id_movie) {
          return UserWatchedMovie.find.where()
                    .and(Expr.eq("id_user_facebook",
id_user_facebook), Expr.eq("id_movie",
id_movie)).findUnique();
     }
```

```
     public static List<UserWatchedMovie>
getUsersWatchedMoviesByPageFacebookId(long id_page_facebook) {
          return
UserWatchedMovie.find.where().eq("id_page_facebook",
id_page_facebook).findList();
     }

     public static List<Movie>
getWatchedMoviesByUserFacebookId(long id_user_facebook) {
          List<Movie> movies = new ArrayList<Movie>();
          for (UserWatchedMovie userWatchedMovie :
getUsersWatchedMoviesByUserFacebookId(id_user_facebook)) {
               movies.add(userWatchedMovie.movie);
          }
          return movies;
     }

     public static List<Movie>
getWatchedMoviesByPageFacebookId(long id_page_facebook) {
          List<Movie> movies = new ArrayList<Movie>();
          for (UserWatchedMovie userWatchedMovie :
getUsersWatchedMoviesByPageFacebookId(id_page_facebook)) {
               movies.add(userWatchedMovie.movie);
          }
          return movies;
     }

     public static boolean hasMovie(long id_user, long
id_movie) {
          for (UserWatchedMovie userWatchedMovie :
getUsersWatchedMoviesByUserFacebookId(id_user)) {
               if (userWatchedMovie.movie.id == id_movie) {
                    return true;
               }
          }
          return false;
     }
}
```

package models

MovieMe/app/models/Actor.java

```
package models;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
```

```
import javax.persistence.OneToMany;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonProperty;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "actors")
public class Actor extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     @JsonIgnore
     public Long id;
     @Constraints.Required
     @JsonProperty("actorName")
     public String name;
     @Constraints.Required
     @JsonProperty("actorId")
     public String id_imdb;

     @OneToMany(mappedBy = "actor")
     public List<MovieActor> movies = new
ArrayList<MovieActor>();
     @OneToMany(mappedBy = "actor")
     public List<UserLikedActor> users = new
ArrayList<UserLikedActor>();

     public static Model.Finder<Long, Actor> find = new
Model.Finder<Long, Actor>(Long.class, Actor.class);

     public Actor() {
          super();
     }

     public Actor(String name, String id_imdb) {
          super();
          this.name = name;
          this.id_imdb = id_imdb;
     }

     public Actor(Long id, String name, String id_imdb) {
```

```
          super();
          this.id = id;
          this.name = name;
          this.id_imdb = id_imdb;
     }

     public Actor(String name, String id_imdb,
List<MovieActor> movies, List<UserLikedActor> users) {
          super();
          this.name = name;
          this.id_imdb = id_imdb;
          this.movies = movies;
          this.users = users;
     }

     public Actor(Long id, String name, String id_imdb,
List<MovieActor> movies, List<UserLikedActor> users) {
          super();
          this.id = id;
          this.name = name;
          this.id_imdb = id_imdb;
          this.movies = movies;
          this.users = users;
     }

     @Override
     public String toString() {
          return "Actor [id=" + id + ", name=" + name + ",
id_imdb=" + id_imdb + ", movies=" + movies + ", users=" +
users
                    + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          Actor other = (Actor) obj;
```

```
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/Country.java

```
package models;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "countries")
public class Country extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     public String name_en;
     public String name_br;

     @OneToMany(mappedBy = "country")
     public List<MovieCountry> movies = new
ArrayList<MovieCountry>();
     @OneToMany(mappedBy = "country")
     public List<UserMovieCountry> users = new
ArrayList<UserMovieCountry>();
```

```
    public static Model.Finder<Long, Country> find = new
Model.Finder<Long, Country>(Long.class, Country.class);

    public Country() {
        super();
    }

    public Country(String name_en) {
        super();
        this.name_en = name_en;
    }

    public Country(Long id, String name_en) {
        super();
        this.id = id;
        this.name_en = name_en;
    }

    public Country(String name_en, String name_br) {
        super();
        this.name_en = name_en;
        this.name_br = name_br;
    }

    public Country(Long id, String name_en, String name_br) {
        super();
        this.id = id;
        this.name_en = name_en;
        this.name_br = name_br;
    }

    public Country(String name_en, String name_br,
List<MovieCountry> movies, List<UserMovieCountry> users) {
        super();
        this.name_en = name_en;
        this.name_br = name_br;
        this.movies = movies;
        this.users = users;
    }

    public Country(Long id, String name_en, String name_br,
List<MovieCountry> movies, List<UserMovieCountry> users) {
        super();
        this.id = id;
        this.name_en = name_en;
        this.name_br = name_br;
        this.movies = movies;
        this.users = users;
    }

    @Override
```

```
     public String toString() {
          return "Country [id=" + id + ", name_en=" + name_en
+ ", name_br=" + name_br + ", movies=" + movies + ", users="
                    + users + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          Country other = (Country) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/Director.java

```
package models;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonProperty;
```

```
import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "directors")
public class Director extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     @JsonIgnore
     public Long id;
     @Constraints.Required
     public String name;
     @Constraints.Required
     @JsonProperty("id")
     public String id_imdb;

     @OneToMany(mappedBy = "director")
     public List<MovieDirector> movies = new
ArrayList<MovieDirector>();
     @OneToMany(mappedBy = "director")
     public List<UserLikedDirector> users = new
ArrayList<UserLikedDirector>();

     public static Model.Finder<Long, Director> find = new
Model.Finder<Long, Director>(Long.class, Director.class);

     public Director() {
          super();
     }

     public Director(String name, String id_imdb) {
          super();
          this.name = name;
          this.id_imdb = id_imdb;
     }

     public Director(Long id, String name, String id_imdb) {
          super();
          this.id = id;
          this.name = name;
          this.id_imdb = id_imdb;
     }
```

```
     public Director(String name, String id_imdb,
List<MovieDirector> movies, List<UserLikedDirector> users) {
          super();
          this.name = name;
          this.id_imdb = id_imdb;
          this.movies = movies;
          this.users = users;
     }

     public Director(Long id, String name, String id_imdb,
List<MovieDirector> movies, List<UserLikedDirector> users) {
          super();
          this.id = id;
          this.name = name;
          this.id_imdb = id_imdb;
          this.movies = movies;
          this.users = users;
     }

     @Override
     public String toString() {
          return "Director [id=" + id + ", name=" + name + ",
id_imdb=" + id_imdb + ", movies=" + movies + ", users="
                    + users + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          Director other = (Director) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }
```

```
}
```

MovieMe/app/models/Evaluation.java

```
package models;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "evaluations")
public class Evaluation extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     public String name_en;
     @Constraints.Required
     public String name_br;

     @OneToMany(mappedBy = "evaluation", cascade =
CascadeType.ALL)
     public List<Recommendation> recommendations = new
ArrayList<Recommendation>();

     public static Model.Finder<Long, Evaluation> find = new
Model.Finder<Long, Evaluation>(Long.class,
               Evaluation.class);

     public Evaluation() {
          super();
     }
```

```
     public Evaluation(String name_en, String name_br,
List<Recommendation> recommendations) {
          super();
          this.name_en = name_en;
          this.name_br = name_br;
          this.recommendations = recommendations;
     }

     public Evaluation(Long id, String name_en, String
name_br, List<Recommendation> recommendations) {
          super();
          this.id = id;
          this.name_en = name_en;
          this.name_br = name_br;
          this.recommendations = recommendations;
     }

     @Override
     public String toString() {
          return "Evaluation [id=" + id + ", name_en=" +
name_en + ", name_br=" + name_br + ", recommendations="
                    + recommendations + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          Evaluation other = (Evaluation) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }
```

```
}
```

MovieMe/app/models/Genre.java

```
package models;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "genres")
public class Genre extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     public String name_en;
     public String name_br;

     @OneToMany(mappedBy = "genre")
     public List<MovieGenre> movies = new
ArrayList<MovieGenre>();
     @OneToMany(mappedBy = "genre")
     public List<UserMovieGenre> users = new
ArrayList<UserMovieGenre>();

     public static Model.Finder<Long, Genre> find = new
Model.Finder<Long, Genre>(Long.class, Genre.class);

     public Genre() {
          super();
     }

     public Genre(String name_en) {
```

```
          super();
          this.name_en = name_en;
     }

     public Genre(Long id, String name_en) {
          super();
          this.id = id;
          this.name_en = name_en;
     }

     public Genre(String name_en, String name_br) {
          super();
          this.name_en = name_en;
          this.name_br = name_br;
     }

     public Genre(Long id, String name_en, String name_br) {
          super();
          this.id = id;
          this.name_en = name_en;
          this.name_br = name_br;
     }

     public Genre(String name_en, String name_br,
List<MovieGenre> movies, List<UserMovieGenre> users) {
          super();
          this.name_en = name_en;
          this.name_br = name_br;
          this.movies = movies;
          this.users = users;
     }

     public Genre(Long id, String name_en, String name_br,
List<MovieGenre> movies, List<UserMovieGenre> users) {
          super();
          this.id = id;
          this.name_en = name_en;
          this.name_br = name_br;
          this.movies = movies;
          this.users = users;
     }

     @Override
     public String toString() {
          return "Genre [id=" + id + ", name_en=" + name_en +
", name_br=" + name_br + ", movies=" + movies + ", users="
                    + users + "]";
     }

     @Override
     public int hashCode() {
```

```
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          Genre other = (Genre) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/Movie.java

```
package models;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.persistence.Transient;

import com.fasterxml.jackson.annotation.JsonProperty;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movies")
public class Movie extends Model {

     private static final long serialVersionUID = 1L;
```

```
     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @JsonProperty("originalTitle")
     public String title_original;
     @JsonProperty("title")
     public String title_br;
     public String title_en;
     public String title_alternative;
     public String title_franchise_original;
     public String title_franchise_br;
     public String title_franchise_alternative;
     public Integer is_movie_franchise;
     public Integer order_franchise;
     @Constraints.Required
     public Integer year;
     public String runtime;
     @JsonProperty("urlPoster")
     public String url_poster;
     @JsonProperty("simplePlot")
     public String plot_en;
     public String plot_br;
     @Constraints.Required
     public Long votes_imdb;
     @Constraints.Required
     public Float rating_imdb;
     @Constraints.Required
     @JsonProperty("idIMDB")
     public String id_imdb;

     // Necessário para construir json resultante do
MyAPIFilms
     @Transient
     public transient String json_type;
     @Transient
     public transient String json_votes_imdb;
     @Transient
     public transient String json_rating_imdb;
     @Transient
     public transient List<MovieAkaJson> json_akas;
     @Transient
     public transient List<String> json_genres;
     @Transient
     public transient List<String> json_countries;
     @Transient
```

```
	public transient List<MovieSimilarJson>
json_movies_similar;
	@Transient
	public transient List<Director> json_directors;
	@Transient
	public transient List<Actor> json_actors;

	@OneToMany(mappedBy = "movie")
	public List<MovieActor> actors = new
ArrayList<MovieActor>();
	@OneToMany(mappedBy = "movie")
	public List<MovieCountry> countries = new
ArrayList<MovieCountry>();
	@OneToMany(mappedBy = "movie")
	public List<MovieDirector> directors = new
ArrayList<MovieDirector>();
	@OneToMany(mappedBy = "movie")
	public List<MovieGenre> genres = new
ArrayList<MovieGenre>();
	@OneToMany(mappedBy = "movie")
	public List<MovieSimilar> movies_similar = new
ArrayList<MovieSimilar>();
	@OneToMany(mappedBy = "movie")
	public List<RecommendedMovie> recommendations = new
ArrayList<RecommendedMovie>();
	@OneToMany(mappedBy = "movie")
	public List<UserLikedRatedMovie> users_liked_rated = new
ArrayList<UserLikedRatedMovie>();
	@OneToMany(mappedBy = "movie")
	public List<UserWatchedMovie> users_watched = new
ArrayList<UserWatchedMovie>();

	public static Model.Finder<Long, Movie> find = new
Model.Finder<Long, Movie>(Long.class, Movie.class);

	public Movie() {
		super();
	}

	public Movie(String title_original, Integer year, Long
votes_imdb, Float rating_imdb, String id_imdb) {
		super();
		this.title_original = title_original;
		this.year = year;
		this.votes_imdb = votes_imdb;
		this.rating_imdb = rating_imdb;
		this.id_imdb = id_imdb;
	}

	public Movie(Long id, String title_original, Integer
year, Long votes_imdb, Float rating_imdb, String id_imdb) {
```

```
          super();
          this.id = id;
          this.title_original = title_original;
          this.year = year;
          this.votes_imdb = votes_imdb;
          this.rating_imdb = rating_imdb;
          this.id_imdb = id_imdb;
     }

     public Movie(String title_original, String title_br,
String title_en, String title_alternative,
               String title_franchise_original, String
title_franchise_br, String title_franchise_alternative,
               Integer is_movie_franchise, Integer
order_franchise, Integer year, String runtime, String
url_poster,
               String plot_en, String plot_br, Long
votes_imdb, Float rating_imdb, String id_imdb) {
          super();
          this.title_original = title_original;
          this.title_br = title_br;
          this.title_en = title_en;
          this.title_alternative = title_alternative;
          this.title_franchise_original =
title_franchise_original;
          this.title_franchise_br = title_franchise_br;
          this.title_franchise_alternative =
title_franchise_alternative;
          this.is_movie_franchise = is_movie_franchise;
          this.order_franchise = order_franchise;
          this.year = year;
          this.runtime = runtime;
          this.url_poster = url_poster;
          this.plot_en = plot_en;
          this.plot_br = plot_br;
          this.votes_imdb = votes_imdb;
          this.rating_imdb = rating_imdb;
          this.id_imdb = id_imdb;
     }

     public Movie(Long id, String title_original, String
title_br, String title_en, String title_alternative,
               String title_franchise_original, String
title_franchise_br, String title_franchise_alternative,
               Integer is_movie_franchise, Integer
order_franchise, Integer year, String runtime, String
url_poster,
               String plot_en, String plot_br, Long
votes_imdb, Float rating_imdb, String id_imdb) {
          super();
          this.id = id;
```

```
            this.title_original = title_original;
            this.title_br = title_br;
            this.title_en = title_en;
            this.title_alternative = title_alternative;
            this.title_franchise_original =
title_franchise_original;
            this.title_franchise_br = title_franchise_br;
            this.title_franchise_alternative =
title_franchise_alternative;
            this.is_movie_franchise = is_movie_franchise;
            this.order_franchise = order_franchise;
            this.year = year;
            this.runtime = runtime;
            this.url_poster = url_poster;
            this.plot_en = plot_en;
            this.plot_br = plot_br;
            this.votes_imdb = votes_imdb;
            this.rating_imdb = rating_imdb;
            this.id_imdb = id_imdb;
        }

        public Movie(String title_original, String title_br,
String title_en, String title_alternative,
                String title_franchise_original, String
title_franchise_br, String title_franchise_alternative,
                Integer is_movie_franchise, Integer
order_franchise, Integer year, String runtime, String
url_poster,
                String plot_en, String plot_br, Long
votes_imdb, Float rating_imdb, String id_imdb,
List<MovieActor> actors,
                List<MovieCountry> countries,
List<MovieDirector> directors, List<MovieGenre> genres,
                List<MovieSimilar> movies_similar,
List<RecommendedMovie> recommendations,
                List<UserLikedRatedMovie> usersLiked,
List<UserWatchedMovie> usersWatched) {
            super();
            this.title_original = title_original;
            this.title_br = title_br;
            this.title_en = title_en;
            this.title_alternative = title_alternative;
            this.title_franchise_original =
title_franchise_original;
            this.title_franchise_br = title_franchise_br;
            this.title_franchise_alternative =
title_franchise_alternative;
            this.is_movie_franchise = is_movie_franchise;
            this.order_franchise = order_franchise;
            this.year = year;
            this.runtime = runtime;
```

```
            this.url_poster = url_poster;
            this.plot_en = plot_en;
            this.plot_br = plot_br;
            this.votes_imdb = votes_imdb;
            this.rating_imdb = rating_imdb;
            this.id_imdb = id_imdb;
            this.actors = actors;
            this.countries = countries;
            this.directors = directors;
            this.genres = genres;
            this.movies_similar = movies_similar;
            this.recommendations = recommendations;
            this.users_liked_rated = usersLiked;
            this.users_watched = usersWatched;
        }

        public Movie(Long id, String title_original, String
title_br, String title_en, String title_alternative,
                String title_franchise_original, String
title_franchise_br, String title_franchise_alternative,
                Integer is_movie_franchise, Integer
order_franchise, Integer year, String runtime, String
url_poster,
                String plot_en, String plot_br, Long
votes_imdb, Float rating_imdb, String id_imdb,
List<MovieActor> actors,
                List<MovieCountry> countries,
List<MovieDirector> directors, List<MovieGenre> genres,
                List<MovieSimilar> movies_similar,
List<RecommendedMovie> recommendations,
                List<UserLikedRatedMovie> usersLiked,
List<UserWatchedMovie> usersWatched) {
            super();
            this.id = id;
            this.title_original = title_original;
            this.title_br = title_br;
            this.title_en = title_en;
            this.title_alternative = title_alternative;
            this.title_franchise_original =
title_franchise_original;
            this.title_franchise_br = title_franchise_br;
            this.title_franchise_alternative =
title_franchise_alternative;
            this.is_movie_franchise = is_movie_franchise;
            this.order_franchise = order_franchise;
            this.year = year;
            this.runtime = runtime;
            this.url_poster = url_poster;
            this.plot_en = plot_en;
            this.plot_br = plot_br;
            this.votes_imdb = votes_imdb;
```

```
            this.rating_imdb = rating_imdb;
            this.id_imdb = id_imdb;
            this.actors = actors;
            this.countries = countries;
            this.directors = directors;
            this.genres = genres;
            this.movies_similar = movies_similar;
            this.recommendations = recommendations;
            this.users_liked_rated = usersLiked;
            this.users_watched = usersWatched;
        }

        public Movie(String title_original, String title_br,
String title_en, String title_alternative,
                String title_franchise_original, String
title_franchise_br, String title_franchise_alternative,
                Integer is_movie_franchise, Integer
order_franchise, Integer year, String runtime, String
url_poster,
                String plot_en, String plot_br, Long
votes_imdb, Float rating_imdb, String id_imdb, String
json_type,
                String json_votes_imdb, String
json_rating_imdb, List<MovieAkaJson> json_akas, List<String>
json_genres,
                List<String> json_countries,
List<MovieSimilarJson> json_movies_similar, List<Director>
json_directors,
                List<Actor> json_actors, List<MovieActor>
actors, List<MovieCountry> countries,
                List<MovieDirector> directors, List<MovieGenre>
genres, List<MovieSimilar> movies_similar,
                List<RecommendedMovie> recommendations,
List<UserLikedRatedMovie> users_liked_rated,
                List<UserWatchedMovie> users_watched) {
            super();
            this.title_original = title_original;
            this.title_br = title_br;
            this.title_en = title_en;
            this.title_alternative = title_alternative;
            this.title_franchise_original =
title_franchise_original;
            this.title_franchise_br = title_franchise_br;
            this.title_franchise_alternative =
title_franchise_alternative;
            this.is_movie_franchise = is_movie_franchise;
            this.order_franchise = order_franchise;
            this.year = year;
            this.runtime = runtime;
            this.url_poster = url_poster;
            this.plot_en = plot_en;
```

```
            this.plot_br = plot_br;
            this.votes_imdb = votes_imdb;
            this.rating_imdb = rating_imdb;
            this.id_imdb = id_imdb;
            this.json_type = json_type;
            this.json_votes_imdb = json_votes_imdb;
            this.json_rating_imdb = json_rating_imdb;
            this.json_akas = json_akas;
            this.json_genres = json_genres;
            this.json_countries = json_countries;
            this.json_movies_similar = json_movies_similar;
            this.json_directors = json_directors;
            this.json_actors = json_actors;
            this.actors = actors;
            this.countries = countries;
            this.directors = directors;
            this.genres = genres;
            this.movies_similar = movies_similar;
            this.recommendations = recommendations;
            this.users_liked_rated = users_liked_rated;
            this.users_watched = users_watched;
        }

        public Movie(Long id, String title_original, String
title_br, String title_en, String title_alternative,
                String title_franchise_original, String
title_franchise_br, String title_franchise_alternative,
                Integer is_movie_franchise, Integer
order_franchise, Integer year, String runtime, String
url_poster,
                String plot_en, String plot_br, Long
votes_imdb, Float rating_imdb, String id_imdb, String
json_type,
                String json_votes_imdb, String
json_rating_imdb, List<MovieAkaJson> json_akas, List<String>
json_genres,
                List<String> json_countries,
List<MovieSimilarJson> json_movies_similar, List<Director>
json_directors,
                List<Actor> json_actors, List<MovieActor>
actors, List<MovieCountry> countries,
                List<MovieDirector> directors, List<MovieGenre>
genres, List<MovieSimilar> movies_similar,
                List<RecommendedMovie> recommendations,
List<UserLikedRatedMovie> users_liked_rated,
                List<UserWatchedMovie> users_watched) {
            super();
            this.id = id;
            this.title_original = title_original;
            this.title_br = title_br;
            this.title_en = title_en;
```

```
          this.title_alternative = title_alternative;
          this.title_franchise_original =
title_franchise_original;
          this.title_franchise_br = title_franchise_br;
          this.title_franchise_alternative =
title_franchise_alternative;
          this.is_movie_franchise = is_movie_franchise;
          this.order_franchise = order_franchise;
          this.year = year;
          this.runtime = runtime;
          this.url_poster = url_poster;
          this.plot_en = plot_en;
          this.plot_br = plot_br;
          this.votes_imdb = votes_imdb;
          this.rating_imdb = rating_imdb;
          this.id_imdb = id_imdb;
          this.json_type = json_type;
          this.json_votes_imdb = json_votes_imdb;
          this.json_rating_imdb = json_rating_imdb;
          this.json_akas = json_akas;
          this.json_genres = json_genres;
          this.json_countries = json_countries;
          this.json_movies_similar = json_movies_similar;
          this.json_directors = json_directors;
          this.json_actors = json_actors;
          this.actors = actors;
          this.countries = countries;
          this.directors = directors;
          this.genres = genres;
          this.movies_similar = movies_similar;
          this.recommendations = recommendations;
          this.users_liked_rated = users_liked_rated;
          this.users_watched = users_watched;
     }

     @Override
     public String toString() {
          return "Movie [id=" + id + ", title_original=" +
title_original + ", title_br=" + title_br + ", title_en="
                    + title_en + ", title_alternative=" +
title_alternative + ", title_franchise_original="
                    + title_franchise_original + ",
title_franchise_br=" + title_franchise_br
                    + ", title_franchise_alternative=" +
title_franchise_alternative + ", is_movie_franchise="
                    + is_movie_franchise + ",
order_franchise=" + order_franchise + ", year=" + year + ",
runtime="
                    + runtime + ", url_poster=" + url_poster +
", plot_en=" + plot_en + ", plot_br=" + plot_br
```

```
                    + ", votes_imdb=" + votes_imdb + ",
rating_imdb=" + rating_imdb + ", id_imdb=" + id_imdb + ",
actors="
                    + actors + ", countries=" + countries + ",
directors=" + directors + ", genres=" + genres
                    + ", movies_similar=" + movies_similar +
", recommendations=" + recommendations + ", usersLiked="
                    + users_liked_rated + ", usersWatched=" +
users_watched + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          Movie other = (Movie) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/MovieActor.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
```

```
import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movies_actors")
public class MovieActor extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_movie")
     public Movie movie;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_actor")
     public Actor actor;

     public static Model.Finder<Long, MovieActor> find = new
Model.Finder<Long, MovieActor>(Long.class,
               MovieActor.class);

     public MovieActor() {
          super();
     }

     public MovieActor(Movie movie, Actor actor) {
          super();
          this.movie = movie;
          this.actor = actor;
     }

     public MovieActor(Long id, Movie movie, Actor actor) {
          super();
          this.id = id;
          this.movie = movie;
          this.actor = actor;
     }

     @Override
     public String toString() {
          return "MovieActor [id=" + id + ", movie=" + movie +
", actor=" + actor + "]";
```

```
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = super.hashCode();
		result = prime * result + ((id == null) ? 0 :
id.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (getClass() != obj.getClass())
			return false;
		MovieActor other = (MovieActor) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

MovieMe/app/models/MovieAkaJson.java

```
package models;

public class MovieAkaJson {

	// Os atributos estão como public e não private porque
esse é o padrão do Play Framework
	// Devido a isso, também não é necessário gerar
getters/setters
	public String country;
	public String title;
	public String comment;

	public MovieAkaJson() {

	}

	public MovieAkaJson(String country, String title, String
comment) {
```

```
            this.country = country;
            this.title = title;
            this.comment = comment;
      }

      @Override
      public String toString() {
            return "MovieAkas [country=" + country + ", title="
+ title + ", comment=" + comment + "]";
      }

}
```

MovieMe/app/models/MovieCountry.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movies_countries")
public class MovieCountry extends Model {

      private static final long serialVersionUID = 1L;

      // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
      // Devido a isso, também não é necessário gerar
getters/setters
      @Id
      @GeneratedValue
      public Long id;
      @Constraints.Required
      @ManyToOne(cascade = CascadeType.ALL)
      @JoinColumn(name = "id_movie")
      public Movie movie;
      @Constraints.Required
      @ManyToOne(cascade = CascadeType.ALL)
      @JoinColumn(name = "id_country")
      public Country country;
```

```
     public static Model.Finder<Long, MovieCountry> find = new
Model.Finder<Long, MovieCountry>(Long.class,
               MovieCountry.class);

     public MovieCountry() {
          super();
     }

     public MovieCountry(Movie movie, Country country) {
          super();
          this.movie = movie;
          this.country = country;
     }

     public MovieCountry(Long id, Movie movie, Country
country) {
          super();
          this.id = id;
          this.movie = movie;
          this.country = country;
     }

     @Override
     public String toString() {
          return "MovieCountry [id=" + id + ", movie=" + movie
+ ", country=" + country + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          MovieCountry other = (MovieCountry) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
```

```
          return true;
     }

}
```

MovieMe/app/models/MovieDirector.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movies_directors")
public class MovieDirector extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_movie")
     public Movie movie;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_director")
     public Director director;

     public static Model.Finder<Long, MovieDirector> find =
new Model.Finder<Long, MovieDirector>(Long.class,
               MovieDirector.class);

     public MovieDirector() {
          super();
     }
```

```
	public MovieDirector(Movie movie, Director director) {
		super();
		this.movie = movie;
		this.director = director;
	}

	public MovieDirector(Long id, Movie movie, Director
director) {
		super();
		this.id = id;
		this.movie = movie;
		this.director = director;
	}

	@Override
	public String toString() {
		return "MovieDirector [id=" + id + ", movie=" +
movie + ", director=" + director + "]";
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = super.hashCode();
		result = prime * result + ((id == null) ? 0 :
id.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (getClass() != obj.getClass())
			return false;
		MovieDirector other = (MovieDirector) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

MovieMe/app/models/MovieGenre.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movies_genres")
public class MovieGenre extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_movie")
     public Movie movie;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_genre")
     public Genre genre;

     public static Model.Finder<Long, MovieGenre> find = new
Model.Finder<Long, MovieGenre>(Long.class,
               MovieGenre.class);

     public MovieGenre() {
          super();
     }

     public MovieGenre(Movie movie, Genre genre) {
          super();
          this.movie = movie;
          this.genre = genre;
     }
```

```
	public MovieGenre(Long id, Movie movie, Genre genre) {
		super();
		this.id = id;
		this.movie = movie;
		this.genre = genre;
	}

	@Override
	public String toString() {
		return "MovieGenre [id=" + id + ", movie=" + movie +
", genre=" + genre + "]";
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = super.hashCode();
		result = prime * result + ((id == null) ? 0 :
id.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (getClass() != obj.getClass())
			return false;
		MovieGenre other = (MovieGenre) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

MovieMe/app/models/MovielensLink.java

```
package models;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

import play.data.validation.Constraints;
```

```
import play.db.ebean.Model;

@Entity
@Table(name = "movielens_links")
public class MovielensLink extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     public Long id_movie;
     @Constraints.Required
     public String id_imdb;
     @Constraints.Required
     public String id_tmdb;

     public static Model.Finder<Long, MovielensLink> find =
new Model.Finder<Long, MovielensLink>(Long.class,
               MovielensLink.class);

     public MovielensLink() {
          super();
     }

     public MovielensLink(Long id_movie, String id_imdb,
String id_tmdb) {
          super();
          this.id_movie = id_movie;
          this.id_imdb = id_imdb;
          this.id_tmdb = id_tmdb;
     }

     @Override
     public String toString() {
          return "MovielensLink [id_movie=" + id_movie + ",
id_imdb=" + id_imdb + ", id_tmdb=" + id_tmdb + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id_movie == null) ? 0 :
id_movie.hashCode());
          return result;
     }

     @Override
```

```
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (getClass() != obj.getClass())
			return false;
		MovielensLink other = (MovielensLink) obj;
		if (id_movie == null) {
			if (other.id_movie != null)
				return false;
		} else if (!id_movie.equals(other.id_movie))
			return false;
		return true;
	}

}
```

MovieMe/app/models/MovielensMovie.java

```
package models;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movielens_movies")
public class MovielensMovie extends Model {

	private static final long serialVersionUID = 1L;

	// Os atributos estão como public e não private porque
esse é o padrão do Play Framework
	// Devido a isso, também não é necessário gerar
getters/setters
	@Id
	public Long id;
	@Constraints.Required
	public String title;
	@Constraints.Required
	public String genres;

	public static Model.Finder<Long, MovielensMovie> find =
new Model.Finder<Long, MovielensMovie>(Long.class,
			MovielensMovie.class);
```

```
	public MovielensMovie() {
		super();
	}

	public MovielensMovie(Long id, String title, String
genres) {
		super();
		this.id = id;
		this.title = title;
		this.genres = genres;
	}

	@Override
	public String toString() {
		return "MovielensMovie [id=" + id + ", title=" +
title + ", genres=" + genres + "]";
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = super.hashCode();
		result = prime * result + ((id == null) ? 0 :
id.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (getClass() != obj.getClass())
			return false;
		MovielensMovie other = (MovielensMovie) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

MovieMe/app/models/MovielensRating.java

```
package models;
```

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movielens_ratings")
public class MovielensRating extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     public Long id_user;
     @Id
     public Long id_movie;
     @Constraints.Required
     public Double rating;

     public static Model.Finder<Long, MovielensRating> find =
new Model.Finder<Long, MovielensRating>(Long.class,
               MovielensRating.class);

     public MovielensRating() {
          super();
     }

     public MovielensRating(Long id_user, Long id_movie,
Double rating) {
          super();
          this.id_user = id_user;
          this.id_movie = id_movie;
          this.rating = rating;
     }

     @Override
     public String toString() {
          return "MovielensRating [id_user=" + id_user + ",
id_movie=" + id_movie + ", rating=" + rating + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
```

```
          result = prime * result + ((id_movie == null) ? 0 :
id_movie.hashCode());
          result = prime * result + ((id_user == null) ? 0 :
id_user.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          MovielensRating other = (MovielensRating) obj;
          if (id_movie == null) {
               if (other.id_movie != null)
                    return false;
          } else if (!id_movie.equals(other.id_movie))
               return false;
          if (id_user == null) {
               if (other.id_user != null)
                    return false;
          } else if (!id_user.equals(other.id_user))
               return false;
          return true;
     }

}
```

MovieMe/app/models/MovieSimilar.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "movies_similar")
public class MovieSimilar extends Model {
```

```
     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_movie")
     public Movie movie;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_movie_similar")
     public Movie movie_similar;

     public static Model.Finder<Long, MovieSimilar> find = new
Model.Finder<Long, MovieSimilar>(Long.class,
               MovieSimilar.class);

     public MovieSimilar() {
          super();
     }

     public MovieSimilar(Movie movie, Movie movie_similar) {
          super();
          this.movie = movie;
          this.movie_similar = movie_similar;
     }

     public MovieSimilar(Long id, Movie movie, Movie
movie_similar) {
          super();
          this.id = id;
          this.movie = movie;
          this.movie_similar = movie_similar;
     }

     @Override
     public String toString() {
          return "MovieSimilar [id=" + id + ", movie=" + movie
+ ", movie_similar=" + movie_similar + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
```

```
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          MovieSimilar other = (MovieSimilar) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/MovieSimilarJson.java

```
package models;

import com.fasterxml.jackson.annotation.JsonProperty;

public class MovieSimilarJson {

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @JsonProperty("id")
     public String id_imdb;

     public MovieSimilarJson() {

     }

     public MovieSimilarJson(String id_imdb) {
          super();
          this.id_imdb = id_imdb;
     }

     @Override
     public String toString() {
```

```
            return "MovieSimilarJson [id_imdb=" + id_imdb + "]";
      }

}
```

MovieMe/app/models/Recommendation.java

```
package models;

import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.OneToOne;
import javax.persistence.Table;
import javax.persistence.Version;

import com.avaje.ebean.annotation.EntityConcurrencyMode;
import com.avaje.ebean.annotation.ConcurrencyMode;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@EntityConcurrencyMode(ConcurrencyMode.NONE)
@Entity
@Table(name = "recommendations")
public class Recommendation extends Model {

      private static final long serialVersionUID = 1L;

      // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
      // Devido a isso, também não é necessário gerar
getters/setters
      @Id
      @GeneratedValue
      public Long id;
      @Constraints.Required
      @OneToOne(cascade = CascadeType.ALL)
      @JoinColumn(name = "id_user_facebook")
      public User user;
      @ManyToOne
      @JoinColumn(name = "id_evaluation")
```

```
     public Evaluation evaluation;
     public String evaluation_opinion;
     @Version
     public Timestamp last_update;

     @OneToMany(mappedBy = "recommendation")
     public List<RecommendedMovie> movies = new
ArrayList<RecommendedMovie>();

     public static Model.Finder<Long, Recommendation> find =
new Model.Finder<Long, Recommendation>(Long.class,
               Recommendation.class);

     public Recommendation() {
          super();
     }

     public Recommendation(User user) {
          super();
          this.user = user;
     }

     public Recommendation(Long id, User user) {
          super();
          this.id = id;
          this.user = user;
     }

     public Recommendation(User user, Evaluation evaluation) {
          super();
          this.user = user;
          this.evaluation = evaluation;
     }

     public Recommendation(Long id, User user, Evaluation
evaluation) {
          super();
          this.id = id;
          this.user = user;
          this.evaluation = evaluation;
     }

     public Recommendation(User user, Evaluation evaluation,
String evaluation_opinion) {
          super();
          this.user = user;
          this.evaluation = evaluation;
          this.evaluation_opinion = evaluation_opinion;
     }
```

```
     public Recommendation(Long id, User user, Evaluation
evaluation, String evaluation_opinion) {
          super();
          this.id = id;
          this.user = user;
          this.evaluation = evaluation;
          this.evaluation_opinion = evaluation_opinion;
     }

     public Recommendation(User user, Evaluation evaluation,
String evaluation_opinion, List<RecommendedMovie> movies) {
          super();
          this.user = user;
          this.evaluation = evaluation;
          this.evaluation_opinion = evaluation_opinion;
          this.movies = movies;
     }

     public Recommendation(Long id, User user, Evaluation
evaluation, String evaluation_opinion,
               List<RecommendedMovie> movies) {
          super();
          this.id = id;
          this.user = user;
          this.evaluation = evaluation;
          this.evaluation_opinion = evaluation_opinion;
          this.movies = movies;
     }

     @Override
     public String toString() {
          return "Recommendation [id=" + id + ", user=" + user
+ ", evaluation=" + evaluation + ", evaluation_opinion="
                    + evaluation_opinion + ", movies=" +
movies + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
```

```
            return false;
        if (getClass() != obj.getClass())
            return false;
        Recommendation other = (Recommendation) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

}
```

MovieMe/app/models/RecommendedMovie.java

```
package models;

import java.sql.Timestamp;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Version;

import com.avaje.ebean.annotation.ConcurrencyMode;
import com.avaje.ebean.annotation.EntityConcurrencyMode;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@EntityConcurrencyMode(ConcurrencyMode.NONE)
@Entity
@Table(name = "recommended_movies")
public class RecommendedMovie extends Model {

    private static final long serialVersionUID = 1L;

    // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
    // Devido a isso, também não é necessário gerar
getters/setters
    @Id
    @GeneratedValue
    public Long id;
    @Constraints.Required
```

```
	@ManyToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "id_recommendation")
	public Recommendation recommendation;
	@Constraints.Required
	@ManyToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "id_movie")
	public Movie movie;
	@Constraints.Required
	public Float score;
	@Constraints.Required
	public Integer approved;
	@Version
	public Timestamp last_update;

	public static Model.Finder<Long, RecommendedMovie> find =
new Model.Finder<Long, RecommendedMovie>(Long.class,
			RecommendedMovie.class);

	public RecommendedMovie() {
		super();
	}

	public RecommendedMovie(Recommendation recommendation,
Movie movie, Float score, Integer approved) {
		super();
		this.recommendation = recommendation;
		this.movie = movie;
		this.score = score;
		this.approved = approved;
	}

	public RecommendedMovie(Long id, Recommendation
recommendation, Movie movie, Float score, Integer approved) {
		super();
		this.id = id;
		this.recommendation = recommendation;
		this.movie = movie;
		this.score = score;
		this.approved = approved;
	}

	@Override
	public String toString() {
		return "RecommendedMovie [id=" + id + ",
recommendation=" + recommendation + ", movie=" + movie + ",
score="
				+ score + ", approved=" + approved + ",
last_update=" + last_update + "]";
	}

	@Override
```

```
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          RecommendedMovie other = (RecommendedMovie) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/Role.java

```
package models;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "roles")
public class Role extends Model {

     private static final long serialVersionUID = 1L;
```

```
     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     public String name_en;
     @Constraints.Required
     public String name_br;

     @OneToMany(mappedBy = "role", cascade = CascadeType.ALL)
     public List<User> users = new ArrayList<User>();

     public static Model.Finder<Long, Role> find = new
Model.Finder<Long, Role>(Long.class, Role.class);

     public Role() {
          super();
     }

     public Role(String name_en, String name_br) {
          super();
          this.name_en = name_en;
          this.name_br = name_br;
     }

     public Role(Long id, String name_en, String name_br) {
          super();
          this.id = id;
          this.name_en = name_en;
          this.name_br = name_br;
     }

     public Role(String name_en, String name_br, List<User>
users) {
          super();
          this.name_en = name_en;
          this.name_br = name_br;
          this.users = users;
     }

     public Role(Long id, String name_en, String name_br,
List<User> users) {
          super();
          this.id = id;
          this.name_en = name_en;
          this.name_br = name_br;
          this.users = users;
     }
```

```
     @Override
     public String toString() {
          return "Role [id=" + id + ", name_en=" + name_en +
", name_br=" + name_br + ", users=" + users + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          Role other = (Role) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/User.java

```
package models;

import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.OneToMany;
import javax.persistence.Table;
```

```
import javax.persistence.Version;

import com.avaje.ebean.annotation.ConcurrencyMode;
import com.avaje.ebean.annotation.EntityConcurrencyMode;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@EntityConcurrencyMode(ConcurrencyMode.NONE)
@Entity
@Table(name = "users")
public class User extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     public Long id_user_facebook;
     @Constraints.Required
     @ManyToOne
     @JoinColumn(name = "id_role")
     public Role role;
     @Constraints.Required
     public String first_name;
     @Constraints.Required
     public String last_name;
     @Constraints.Required
     public Integer total_liked_movies;
     @Constraints.Required
     public Integer total_watched_movies;
     @Constraints.Required
     public Integer total_rated_movies;
     @Constraints.Required
     public Integer total_liked_actors_directors;
     @Version
     public Timestamp last_update;

     @OneToMany(mappedBy = "user")
     public List<UserLikedActor> liked_actors = new
ArrayList<UserLikedActor>();
     @OneToMany(mappedBy = "user")
     public List<UserLikedDirector> liked_directors = new
ArrayList<UserLikedDirector>();
     @OneToMany(mappedBy = "user")
     public List<UserLikedRatedMovie> liked_rated_movies = new
ArrayList<UserLikedRatedMovie>();
     @OneToMany(mappedBy = "user")
```

```
     public List<UserWatchedMovie> watched_movies = new
ArrayList<UserWatchedMovie>();
     @OneToMany(mappedBy = "user")
     public List<UserMovieCountry> movies_countries = new
ArrayList<UserMovieCountry>();
     @OneToMany(mappedBy = "user")
     public List<UserMovieGenre> movies_genres = new
ArrayList<UserMovieGenre>();
     @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
     public List<UserLike> likes = new ArrayList<UserLike>();

     public static Model.Finder<Long, User> find = new
Model.Finder<Long, User>(Long.class, User.class);

     public User() {
          super();
     }

     public User(Role role, String first_name, String
last_name, Integer total_liked_movies,
               Integer total_watched_movies, Integer
total_rated_movies, Integer total_liked_actors_directors) {
          super();
          this.role = role;
          this.first_name = first_name;
          this.last_name = last_name;
          this.total_liked_movies = total_liked_movies;
          this.total_watched_movies = total_watched_movies;
          this.total_rated_movies = total_rated_movies;
          this.total_liked_actors_directors =
total_liked_actors_directors;
     }

     public User(Long id_user_facebook, Role role, String
first_name, String last_name, Integer total_liked_movies,
               Integer total_watched_movies, Integer
total_rated_movies, Integer total_liked_actors_directors) {
          super();
          this.id_user_facebook = id_user_facebook;
          this.role = role;
          this.first_name = first_name;
          this.last_name = last_name;
          this.total_liked_movies = total_liked_movies;
          this.total_watched_movies = total_watched_movies;
          this.total_rated_movies = total_rated_movies;
          this.total_liked_actors_directors =
total_liked_actors_directors;
     }

     public User(Role role, String first_name, String
last_name, Integer total_liked_movies,
```

```
                Integer total_watched_movies, Integer
total_rated_movies, Integer total_liked_actors_directors,
                Timestamp last_update) {
            super();
            this.role = role;
            this.first_name = first_name;
            this.last_name = last_name;
            this.total_liked_movies = total_liked_movies;
            this.total_watched_movies = total_watched_movies;
            this.total_rated_movies = total_rated_movies;
            this.total_liked_actors_directors =
total_liked_actors_directors;
            this.last_update = last_update;
        }

        public User(Long id_user_facebook, Role role, String
first_name, String last_name, Integer total_liked_movies,
                Integer total_watched_movies, Integer
total_rated_movies, Integer total_liked_actors_directors,
                Timestamp last_update) {
            super();
            this.id_user_facebook = id_user_facebook;
            this.role = role;
            this.first_name = first_name;
            this.last_name = last_name;
            this.total_liked_movies = total_liked_movies;
            this.total_watched_movies = total_watched_movies;
            this.total_rated_movies = total_rated_movies;
            this.total_liked_actors_directors =
total_liked_actors_directors;
            this.last_update = last_update;
        }

        public User(Role role, String first_name, String
last_name, Integer total_liked_movies,
                Integer total_watched_movies, Integer
total_rated_movies, Integer total_liked_actors_directors,
                Timestamp last_update, List<UserLikedActor>
liked_actors, List<UserLikedDirector> liked_directors,
                List<UserLikedRatedMovie> liked_rated_movies,
List<UserWatchedMovie> watched_movies,
                List<UserMovieCountry> movies_countries,
List<UserMovieGenre> movies_genres, List<UserLike> likes) {
            super();
            this.role = role;
            this.first_name = first_name;
            this.last_name = last_name;
            this.total_liked_movies = total_liked_movies;
            this.total_watched_movies = total_watched_movies;
            this.total_rated_movies = total_rated_movies;
```

```
          this.total_liked_actors_directors =
total_liked_actors_directors;
          this.last_update = last_update;
          this.liked_actors = liked_actors;
          this.liked_directors = liked_directors;
          this.liked_rated_movies = liked_rated_movies;
          this.watched_movies = watched_movies;
          this.movies_countries = movies_countries;
          this.movies_genres = movies_genres;
          this.likes = likes;
     }

     public User(Long id_user_facebook, Role role, String
first_name, String last_name, Integer total_liked_movies,
               Integer total_watched_movies, Integer
total_rated_movies, Integer total_liked_actors_directors,
               Timestamp last_update, List<UserLikedActor>
liked_actors, List<UserLikedDirector> liked_directors,
               List<UserLikedRatedMovie> liked_rated_movies,
List<UserWatchedMovie> watched_movies,
               List<UserMovieCountry> movies_countries,
List<UserMovieGenre> movies_genres, List<UserLike> likes) {
          super();
          this.id_user_facebook = id_user_facebook;
          this.role = role;
          this.first_name = first_name;
          this.last_name = last_name;
          this.total_liked_movies = total_liked_movies;
          this.total_watched_movies = total_watched_movies;
          this.total_rated_movies = total_rated_movies;
          this.total_liked_actors_directors =
total_liked_actors_directors;
          this.last_update = last_update;
          this.liked_actors = liked_actors;
          this.liked_directors = liked_directors;
          this.liked_rated_movies = liked_rated_movies;
          this.watched_movies = watched_movies;
          this.movies_countries = movies_countries;
          this.movies_genres = movies_genres;
          this.likes = likes;
     }

     @Override
     public String toString() {
          return "User [id_user_facebook=" + id_user_facebook
+ ", role=" + role + ", first_name=" + first_name
                    + ", last_name=" + last_name + ",
total_liked_movies=" + total_liked_movies + ",
total_watched_movies="
                    + total_watched_movies + ",
total_rated_movies=" + total_rated_movies
```

```
                    + ", total_liked_actors_directors=" +
total_liked_actors_directors + ", last_update=" + last_update
                    + ", liked_actors=" + liked_actors + ",
liked_directors=" + liked_directors + ", liked_rated_movies="
                    + liked_rated_movies + ", watched_movies="
+ watched_movies + ", movies_countries=" + movies_countries
                    + ", movies_genres=" + movies_genres + ",
likes=" + likes + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id_user_facebook ==
null) ? 0 : id_user_facebook.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          User other = (User) obj;
          if (id_user_facebook == null) {
               if (other.id_user_facebook != null)
                    return false;
          } else if
(!id_user_facebook.equals(other.id_user_facebook))
               return false;
          return true;
     }

}
```

MovieMe/app/models/UserLike.java

```
package models;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;
```

```
import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "users_likes")
public class UserLike extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne
     @JoinColumn(name = "id_user_facebook")
     public User user;
     @Constraints.Required
     public Long id_page_facebook;
     @Constraints.Required
     public String name_page_facebook;
     @Constraints.Required
     public String category_page_facebook;

     public static Model.Finder<Long, UserLike> find = new
Model.Finder<Long, UserLike>(Long.class, UserLike.class);

     public UserLike() {
          super();
     }

     public UserLike(User user, Long id_page_facebook, String
name_page_facebook, String category_page_facebook) {
          super();
          this.user = user;
          this.id_page_facebook = id_page_facebook;
          this.name_page_facebook = name_page_facebook;
          this.category_page_facebook =
category_page_facebook;
     }

     public UserLike(Long id, User user, Long
id_page_facebook, String name_page_facebook,
               String category_page_facebook) {
          super();
          this.id = id;
          this.user = user;
          this.id_page_facebook = id_page_facebook;
```

```
          this.name_page_facebook = name_page_facebook;
          this.category_page_facebook =
category_page_facebook;
     }

     @Override
     public String toString() {
          return "UserLike [id=" + id + ", user=" + user + ",
id_page_facebook=" + id_page_facebook
                    + ", name_page_facebook=" +
name_page_facebook + ", category_page_facebook=" +
category_page_facebook
                    + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          UserLike other = (UserLike) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/UserLikedActor.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "users_liked_actors")
public class UserLikedActor extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_user_facebook")
     public User user;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_actor")
     public Actor actor;
     @Constraints.Required
     public Long id_page_facebook;

     public static Model.Finder<Long, UserLikedActor> find =
new Model.Finder<Long, UserLikedActor>(Long.class,
               UserLikedActor.class);

     public UserLikedActor() {
          super();
     }

     public UserLikedActor(User user, Actor actor, Long
id_page_facebook) {
          super();
          this.user = user;
          this.actor = actor;
          this.id_page_facebook = id_page_facebook;
     }

     public UserLikedActor(Long id, User user, Actor actor,
Long id_page_facebook) {
          super();
```

```
          this.id = id;
          this.user = user;
          this.actor = actor;
          this.id_page_facebook = id_page_facebook;
     }

     @Override
     public String toString() {
          return "UserLikedActor [id=" + id + ", user=" + user
+ ", actor=" + actor + ", id_page_facebook="
                    + id_page_facebook + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          UserLikedActor other = (UserLikedActor) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/UserLikedDirector.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
```

```
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "users_liked_directors")
public class UserLikedDirector extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_user_facebook")
     public User user;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_director")
     public Director director;
     @Constraints.Required
     public Long id_page_facebook;

     public static Model.Finder<Long, UserLikedDirector> find
= new Model.Finder<Long, UserLikedDirector>(Long.class,
               UserLikedDirector.class);

     public UserLikedDirector() {
          super();
     }

     public UserLikedDirector(User user, Director director,
Long id_page_facebook) {
          super();
          this.user = user;
          this.director = director;
          this.id_page_facebook = id_page_facebook;
     }

     public UserLikedDirector(Long id, User user, Director
director, Long id_page_facebook) {
          super();
          this.id = id;
          this.user = user;
```

```
          this.director = director;
          this.id_page_facebook = id_page_facebook;
     }

     @Override
     public String toString() {
          return "UserLikedDirector [id=" + id + ", user=" +
user + ", director=" + director + ", id_page_facebook="
                    + id_page_facebook + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          UserLikedDirector other = (UserLikedDirector) obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/UserLikedRatedMovie.java

```
package models;

import java.sql.Timestamp;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
```

```
import javax.persistence.ManyToOne;
import javax.persistence.Table;
import javax.persistence.Version;

import com.avaje.ebean.annotation.ConcurrencyMode;
import com.avaje.ebean.annotation.EntityConcurrencyMode;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@EntityConcurrencyMode(ConcurrencyMode.NONE)
@Entity
@Table(name = "users_liked_rated_movies")
public class UserLikedRatedMovie extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_user_facebook")
     public User user;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_movie")
     public Movie movie;
     @Constraints.Required
     public Long id_page_facebook;
     @Constraints.Required
     public Float rating;
     @Version
     public Timestamp last_update;

     public static Model.Finder<Long, UserLikedRatedMovie>
find = new Model.Finder<Long, UserLikedRatedMovie>(Long.class,
               UserLikedRatedMovie.class);

     public UserLikedRatedMovie() {
          super();
     }

     public UserLikedRatedMovie(User user, Movie movie, Long
id_page_facebook, Float rating) {
          super();
          this.user = user;
```

```
          this.movie = movie;
          this.id_page_facebook = id_page_facebook;
          this.rating = rating;
     }

     public UserLikedRatedMovie(Long id, User user, Movie
movie, Long id_page_facebook, Float rating) {
          super();
          this.id = id;
          this.user = user;
          this.movie = movie;
          this.id_page_facebook = id_page_facebook;
          this.rating = rating;
     }

     public UserLikedRatedMovie(User user, Movie movie, Long
id_page_facebook, Float rating, Timestamp last_update) {
          super();
          this.user = user;
          this.movie = movie;
          this.id_page_facebook = id_page_facebook;
          this.rating = rating;
          this.last_update = last_update;
     }

     public UserLikedRatedMovie(Long id, User user, Movie
movie, Long id_page_facebook, Float rating,
                Timestamp last_update) {
          super();
          this.id = id;
          this.user = user;
          this.movie = movie;
          this.id_page_facebook = id_page_facebook;
          this.rating = rating;
          this.last_update = last_update;
     }

     @Override
     public String toString() {
          return "UserLikedRatedMovie [id=" + id + ", user=" +
user + ", movie=" + movie + ", id_page_facebook="
                    + id_page_facebook + ", rating=" + rating
+ ", last_update=" + last_update + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
```

```
          return result;
     }

     @Override
     public boolean equals(Object obj) {
          if (this == obj)
               return true;
          if (!super.equals(obj))
               return false;
          if (getClass() != obj.getClass())
               return false;
          UserLikedRatedMovie other = (UserLikedRatedMovie)
obj;
          if (id == null) {
               if (other.id != null)
                    return false;
          } else if (!id.equals(other.id))
               return false;
          return true;
     }

}
```

MovieMe/app/models/UserMovieCountry.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "users_movies_countries")
public class UserMovieCountry extends Model {

     private static final long serialVersionUID = 1L;

     // Os atributos estão como public e não private porque
esse é o padrão do Play Framework
     // Devido a isso, também não é necessário gerar
getters/setters
     @Id
     @GeneratedValue
```

```
     public Long id;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_user_facebook")
     public User user;
     @Constraints.Required
     @ManyToOne(cascade = CascadeType.ALL)
     @JoinColumn(name = "id_country")
     public Country country;
     @Constraints.Required
     public Integer quantity;

     public static Model.Finder<Long, UserMovieCountry> find =
new Model.Finder<Long, UserMovieCountry>(Long.class,
               UserMovieCountry.class);

     public UserMovieCountry() {
          super();
     }

     public UserMovieCountry(User user, Country country,
Integer quantity) {
          super();
          this.user = user;
          this.country = country;
          this.quantity = quantity;
     }

     public UserMovieCountry(Long id, User user, Country
country, Integer quantity) {
          super();
          this.id = id;
          this.user = user;
          this.country = country;
          this.quantity = quantity;
     }

     @Override
     public String toString() {
          return "UserMovieCountry [id=" + id + ", user=" +
user + ", country=" + country + ", quantity=" + quantity
                    + "]";
     }

     @Override
     public int hashCode() {
          final int prime = 31;
          int result = super.hashCode();
          result = prime * result + ((id == null) ? 0 :
id.hashCode());
          return result;
```

```
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (getClass() != obj.getClass())
			return false;
		UserMovieCountry other = (UserMovieCountry) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

MovieMe/app/models/UserMovieGenre.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "users_movies_genres")
public class UserMovieGenre extends Model {

	private static final long serialVersionUID = 1L;

	// Os atributos estão como public e não private porque
esse é o padrão do Play Framework
	// Devido a isso, também não é necessário gerar
getters/setters
	@Id
	@GeneratedValue
	public Long id;
	@Constraints.Required
```

```
	@ManyToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "id_user_facebook")
	public User user;
	@Constraints.Required
	@ManyToOne(cascade = CascadeType.ALL)
	@JoinColumn(name = "id_genre")
	public Genre genre;
	@Constraints.Required
	public Integer quantity;

	public static Model.Finder<Long, UserMovieGenre> find =
new Model.Finder<Long, UserMovieGenre>(Long.class,
			UserMovieGenre.class);

	public UserMovieGenre() {
		super();
	}

	public UserMovieGenre(User user, Genre genre, Integer
quantity) {
		super();
		this.user = user;
		this.genre = genre;
		this.quantity = quantity;
	}

	public UserMovieGenre(Long id, User user, Genre genre,
Integer quantity) {
		super();
		this.id = id;
		this.user = user;
		this.genre = genre;
		this.quantity = quantity;
	}

	@Override
	public String toString() {
		return "UserMovieGenre [id=" + id + ", user=" + user
+ ", genre=" + genre + ", quantity=" + quantity + "]";
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = super.hashCode();
		result = prime * result + ((id == null) ? 0 :
id.hashCode());
		return result;
	}

	@Override
```

```
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (!super.equals(obj))
			return false;
		if (getClass() != obj.getClass())
			return false;
		UserMovieGenre other = (UserMovieGenre) obj;
		if (id == null) {
			if (other.id != null)
				return false;
		} else if (!id.equals(other.id))
			return false;
		return true;
	}

}
```

MovieMe/app/models/UserWatchedMovie.java

```
package models;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import play.data.validation.Constraints;
import play.db.ebean.Model;

@Entity
@Table(name = "users_watched_movies")
public class UserWatchedMovie extends Model {

	private static final long serialVersionUID = 1L;

	// Os atributos estão como public e não private porque
esse é o padrão do Play Framework
	// Devido a isso, também não é necessário gerar
getters/setters
	@Id
	@GeneratedValue
	public Long id;
	@Constraints.Required
	@ManyToOne(optional = true, cascade = CascadeType.ALL)
	@JoinColumn(name = "id_user_facebook")
	public User user;
```

```
	@Constraints.Required
	@ManyToOne(optional = true, cascade = CascadeType.ALL)
	@JoinColumn(name = "id_movie")
	public Movie movie;
	@Constraints.Required
	public Long id_page_facebook;

	public static Model.Finder<Long, UserWatchedMovie> find =
new Model.Finder<Long, UserWatchedMovie>(Long.class,
			UserWatchedMovie.class);

	public UserWatchedMovie() {
		super();
	}

	public UserWatchedMovie(User user, Movie movie, Long
id_page_facebook) {
		super();
		this.user = user;
		this.movie = movie;
		this.id_page_facebook = id_page_facebook;
	}

	public UserWatchedMovie(Long id, User user, Movie movie,
Long id_page_facebook) {
		super();
		this.id = id;
		this.user = user;
		this.movie = movie;
		this.id_page_facebook = id_page_facebook;
	}

	@Override
	public String toString() {
		return "UserWatchedMovie [id=" + id + ", user=" +
user + ", movie=" + movie + ", id_page_facebook="
				+ id_page_facebook + "]";
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = super.hashCode();
		result = prime * result + ((id == null) ? 0 :
id.hashCode());
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
```

```
            return true;
        if (!super.equals(obj))
            return false;
        if (getClass() != obj.getClass())
            return false;
        UserWatchedMovie other = (UserWatchedMovie) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

}
```

package recommender

MovieMe/app/recommender/ActorRescorer.java

```
package recommender;

import java.util.Collections;
import java.util.List;

import org.apache.mahout.cf.taste.recommender.IDRescorer;

import crud.MovieCRUD;
import models.Actor;

public class ActorRescorer implements IDRescorer {

    private final List<Actor> likedActors;

    public ActorRescorer(List<Actor> likedActors) {
        // Lista com atores curtidos pelo usuário
        this.likedActors = likedActors;
    }

    @Override
    public double rescore(long currentMovieId, double
originalScore) {
        // Aumenta o peso em 20% dos filmes que contêm os
atores curtidos pelo usuário
        if
(!Collections.disjoint(MovieCRUD.getActorsByMovieId(currentMov
ieId), likedActors)) {
            return originalScore * 1.2;
        }
        return originalScore;
    }
```

```
     @Override
     public boolean isFiltered(long currentMovieId) {
          return false;
     }

}
```

MovieMe/app/recommender/CountryRescorer.java

```
package recommender;

import java.util.Collections;
import java.util.List;

import org.apache.mahout.cf.taste.recommender.IDRescorer;

import crud.MovieCRUD;
import models.Country;

public class CountryRescorer implements IDRescorer {

     private final List<Country> topLikedMoviesCountries;

     public CountryRescorer(List<Country>
topLikedMoviesCountries) {
          // Lista com os países de origem mais recorrentes
dos filmes curtidos/assistidos/avaliados pelo usuário
          this.topLikedMoviesCountries =
topLikedMoviesCountries;
     }

     @Override
     public double rescore(long currentMovieId, double
originalScore) {
          // Aumenta o peso em 10% dos filmes que são dos
países de origem mais recorrentes
          if
(!Collections.disjoint(MovieCRUD.getCountriesByMovieId(current
MovieId), topLikedMoviesCountries)) {
               return originalScore * 1.1;
          }
          return originalScore;
     }

     @Override
     public boolean isFiltered(long currentMovieId) {
          return false;
     }
```

```
}
```

MovieMe/app/recommender/DirectorRescorer.java

```
package recommender;

import java.util.Collections;
import java.util.List;

import org.apache.mahout.cf.taste.recommender.IDRescorer;

import crud.MovieCRUD;
import models.Director;

public class DirectorRescorer implements IDRescorer {

     private final List<Director> likedDirectors;

     public DirectorRescorer(List<Director> likedDirectors) {
          // Lista com diretores curtidos pelo usuário
          this.likedDirectors = likedDirectors;
     }

     @Override
     public double rescore(long currentMovieId, double
originalScore) {
          // Aumenta o peso em 20% dos filmes que contêm os
diretores curtidos pelo usuário
          if
(!Collections.disjoint(MovieCRUD.getDirectorsByMovieId(current
MovieId), likedDirectors)) {
               return originalScore * 1.2;
          }
          return originalScore;
     }

     @Override
     public boolean isFiltered(long currentMovieId) {
          return false;
     }

}
```

MovieMe/app/recommender/Evaluator.java

```
package recommender;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
```

```
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import org.apache.mahout.cf.taste.eval.IRStatistics;
import org.apache.mahout.cf.taste.eval.RecommenderBuilder;
import org.apache.mahout.cf.taste.eval.RecommenderEvaluator;
import
org.apache.mahout.cf.taste.eval.RecommenderIRStatsEvaluator;
import
org.apache.mahout.cf.taste.impl.eval.AverageAbsoluteDifference
RecommenderEvaluator;
import
org.apache.mahout.cf.taste.impl.eval.GenericRecommenderIRStats
Evaluator;
import
org.apache.mahout.cf.taste.impl.eval.RMSRecommenderEvaluator;
import
org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import
org.apache.mahout.cf.taste.impl.neighborhood.NearestNUserNeigh
borhood;
import
org.apache.mahout.cf.taste.impl.neighborhood.ThresholdUserNeig
hborhood;
import
org.apache.mahout.cf.taste.impl.recommender.GenericBooleanPref
UserBasedRecommender;
import
org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRe
commender;
import
org.apache.mahout.cf.taste.impl.similarity.EuclideanDistanceSi
milarity;
import
org.apache.mahout.cf.taste.impl.similarity.LogLikelihoodSimila
rity;
import
org.apache.mahout.cf.taste.impl.similarity.PearsonCorrelationS
imilarity;
import org.apache.mahout.cf.taste.model.DataModel;
import
org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.cf.taste.similarity.UserSimilarity;
import org.apache.mahout.common.RandomUtils;

public class Evaluator {

     private final DataModel model;

     public Evaluator() throws TasteException, IOException {
```

```
          // Garante que os mesmo dados aleatórios serão
utilizados toda vez que é executado um teste
          RandomUtils.useTestSeed();
          // Arquivo .csv com as avaliações de filmes dos
usuários do movielens - com notas
          this.model = new FileDataModel(new
File("data/movielens/movielens_ratings_complete_rated_data.csv
"));
          // Arquivo .csv com as avaliações de filmes dos
usuários do movielens - sem notas
          //this.model = new FileDataModel(new
File("data/movielens/movielens_ratings_complete_boolean_data.c
sv"));
     }

     public double getAverageAbsoluteDifferenceEvaluation()
throws TasteException, IOException {
          // MAE (Mean Absolute Error)
          RecommenderEvaluator
averageAbsoluteDifferenceEvaluator = new
AverageAbsoluteDifferenceRecommenderEvaluator();

          RecommenderBuilder recommenderBuilder = new
RecommenderBuilder() {
               @Override
               public Recommender buildRecommender(DataModel
model) throws TasteException {
                    // Métrica de similaridade - Distância
euclidiana
                    UserSimilarity euclideanDistanceSimilarity
= new EuclideanDistanceSimilarity(model);
                    // Métrica de similaridade - Correlação de
Pearson
                    UserSimilarity
pearsonCorrelationSimilarity = new
PearsonCorrelationSimilarity(model);

                    // Definição da vizinha - Nearest N User
Neighborhood
                    UserNeighborhood nearestNUserNeighborhood
= new NearestNUserNeighborhood(200,
                              euclideanDistanceSimilarity,
model);
                    // Definição da vizinha - Threshold User
Neighborhood
                    UserNeighborhood thresholdUserNeighborhood
= new ThresholdUserNeighborhood(0.9,
                              euclideanDistanceSimilarity,
model);
```

```
                    return new
GenericUserBasedRecommender(model, nearestNUserNeighborhood,
euclideanDistanceSimilarity);
                }
            };
            // 0.7 indica que apenas 30% dos dados serão
utilizados para testes
            // 1.0 indica a porcentagem de dados que serão
utilizadas, nesse caso, 100%
            double score =
averageAbsoluteDifferenceEvaluator.evaluate(recommenderBuilder
, null, model, 0.7, 1.0);
            return score;
        }

        public double getRMSEvaluation() throws TasteException,
IOException {
            //RMSE (Root Mean Square Error)
            RecommenderEvaluator rmsEvaluator = new
RMSRecommenderEvaluator();

            RecommenderBuilder recommenderBuilder = new
RecommenderBuilder() {
                @Override
                public Recommender buildRecommender(DataModel
model) throws TasteException {
                    // Métrica de similaridade - Distância
euclidiana
                    UserSimilarity euclideanDistanceSimilarity
= new EuclideanDistanceSimilarity(model);
                    // Métrica de similaridade - Correlação de
Pearson
                    UserSimilarity
pearsonCorrelationSimilarity = new
PearsonCorrelationSimilarity(model);

                    // Definição da vizinha - Nearest N User
Neighborhood
                    UserNeighborhood nearestNUserNeighborhood
= new NearestNUserNeighborhood(200,
                            euclideanDistanceSimilarity,
model);
                    // Definição da vizinha - Threshold User
Neighborhood
                    UserNeighborhood thresholdUserNeighborhood
= new ThresholdUserNeighborhood(0.9,
                            euclideanDistanceSimilarity,
model);
                    return new
GenericUserBasedRecommender(model, nearestNUserNeighborhood,
euclideanDistanceSimilarity);
```

```
                }
            };
            // 0.7 indica que apenas 30% dos dados serão
utilizados para testes
            // 1.0 indica a porcentagem de dados que serão
utilizadas, nesse caso, 100%
            double score =
rmsEvaluator.evaluate(recommenderBuilder, null, model, 0.7,
1.0);
            return score;
        }

        public List<Double> getPrecisionAndRecallEvaluation()
throws TasteException, IOException {
            // Precisão e Revocação (Precision and Recall)
            RecommenderIRStatsEvaluator irStatsEvaluator = new
GenericRecommenderIRStatsEvaluator();

            RecommenderBuilder recommenderBuilder = new
RecommenderBuilder() {
                @Override
                public Recommender buildRecommender(DataModel
model) throws TasteException {
                    // Métrica de similaridade - LogLikelihood
                    UserSimilarity logLikelihoodSimilarity =
new LogLikelihoodSimilarity(model);

                    // Definição da vizinha - Nearest N User
Neighborhood
                    UserNeighborhood nearestNUserNeighborhood
= new NearestNUserNeighborhood(10, logLikelihoodSimilarity,
                            model);
                    // Definição da vizinha - Threshold User
Neighborhood
                    UserNeighborhood thresholdUserNeighborhood
= new ThresholdUserNeighborhood(0.7, logLikelihoodSimilarity,
                            model);
                    return new
GenericBooleanPrefUserBasedRecommender(model,
nearestNUserNeighborhood,
                            logLikelihoodSimilarity);
                }
            };

            IRStatistics stats =
irStatsEvaluator.evaluate(recommenderBuilder, null, model,
null, 10,

        GenericRecommenderIRStatsEvaluator.CHOOSE_THRESHOLD,
1.0);
            List<Double> scores = new ArrayList<Double>();
```

```
          scores.add(stats.getPrecision());
          scores.add(stats.getRecall());
          return scores;
     }

}
```

MovieMe/app/recommender/GenreRescorer.java

```
package recommender;

import java.util.Collections;
import java.util.List;

import org.apache.mahout.cf.taste.recommender.IDRescorer;

import crud.MovieCRUD;
import models.Genre;

public class GenreRescorer implements IDRescorer {

     private final List<Genre> topLikedMoviesGenres;

     public GenreRescorer(List<Genre> topLikedMoviesGenres) {
          // Lista com os gêneros mais recorrentes dos filmes
curtidos/assistidos/avaliados pelo usuário
          this.topLikedMoviesGenres = topLikedMoviesGenres;
     }

     @Override
     public double rescore(long currentMovieId, double
originalScore) {
          // Aumenta o peso em 20% dos filmes que possuem os
gêneros mais recorrentes
          if
(!Collections.disjoint(MovieCRUD.getGenresByMovieId(currentMov
ieId), topLikedMoviesGenres)) {
               return originalScore * 1.2;
          }
          return originalScore;
     }

     @Override
     public boolean isFiltered(long currentMovieId) {
          return false;
     }

}
```

MovieMe/app/recommender/MovieRatingRescorer.java

```
package recommender;

import org.apache.mahout.cf.taste.recommender.IDRescorer;

import crud.MovieCRUD;
import models.Movie;

public class MovieRatingRescorer implements IDRescorer {

     private final float likedMoviesWithMinimumRating;

     public MovieRatingRescorer(float minimumRating) {
          // Lista com os filmes curtidos/assistidos/avaliados
pelo usuário que tem a nota do imdb mínima
          this.likedMoviesWithMinimumRating = minimumRating;
     }

     @Override
     public double rescore(long currentMovieId, double
originalScore) {
          // Aumenta o peso em 10% dos filmes que contêm a
nota do imdb mínima
          Movie currentMovie =
MovieCRUD.getMovieById(currentMovieId);
          if (currentMovie != null) {
               if (currentMovie.rating_imdb >=
likedMoviesWithMinimumRating) {
                    return originalScore * 1.1;
               }
          }
          return originalScore;
     }

     @Override
     public boolean isFiltered(long currentMovieId) {
          return false;
     }

}
```

MovieMe/app/recommender/MovieRecommender.java

```
package recommender;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
```

```
import java.util.List;
import java.util.Set;

import org.apache.mahout.cf.taste.common.TasteException;
import
org.apache.mahout.cf.taste.impl.model.GenericUserPreferenceArr
ay;
import
org.apache.mahout.cf.taste.impl.model.PlusAnonymousUserDataMod
el;
import
org.apache.mahout.cf.taste.impl.model.file.FileDataModel;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.model.PreferenceArray;
import org.apache.mahout.cf.taste.recommender.IDRescorer;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;

import crud.MovieCRUD;
import crud.UserCRUD;
import crud.UserMovieCountryCRUD;
import crud.UserMovieGenreCRUD;

import
org.apache.mahout.cf.taste.impl.model.GenericBooleanPrefDataMo
del;

import models.Movie;
import models.User;
import models.UserLikedRatedMovie;
import models.UserWatchedMovie;

public class MovieRecommender {

     private final User user;

     public MovieRecommender(User user) {
          this.user = user;
     }

     public List<RecommendedItem> getRecommendations() throws
TasteException, IOException {
          int userTotalOfLikedAndRatedMovies =
user.liked_rated_movies.size();
          int userTotalOfWatchedMovies =
user.watched_movies.size();
          int
numberOfRecommendedItemsForRecommenderWithRatedData = 0;
          int
numberOfRecommendedItemsForRecommenderWithBooleanData = 0;
```

```
          // Verifica se existem mais filmes
curtidos/avaliados ou assistidos
          // Necessário para definir a quantidade de filmes
que serão recomendados por cada cálculo de recomendação
          if (userTotalOfLikedAndRatedMovies > 0 &&
userTotalOfWatchedMovies > 0) {
               if (userTotalOfLikedAndRatedMovies >
userTotalOfWatchedMovies) {

     numberOfRecommendedItemsForRecommenderWithRatedData = 7;

     numberOfRecommendedItemsForRecommenderWithBooleanData =
3;
               } else if (userTotalOfLikedAndRatedMovies <
userTotalOfWatchedMovies) {

     numberOfRecommendedItemsForRecommenderWithRatedData = 3;

     numberOfRecommendedItemsForRecommenderWithBooleanData =
7;
               } else {

     numberOfRecommendedItemsForRecommenderWithRatedData = 5;

     numberOfRecommendedItemsForRecommenderWithBooleanData =
5;

               }
          } else if (userTotalOfLikedAndRatedMovies > 0 &&
userTotalOfWatchedMovies == 0) {

     numberOfRecommendedItemsForRecommenderWithRatedData = 10;
          } else if (userTotalOfLikedAndRatedMovies == 0 &&
userTotalOfWatchedMovies > 0) {

     numberOfRecommendedItemsForRecommenderWithBooleanData =
10;
          }

          // Lista que junta os filmes recomendados dos
diferentes cálculos de recomendação
          List<RecommendedItem> allRecommendedItems = new
ArrayList<RecommendedItem>();

          // Se houverem filmes curtidos/avaliados, o cálculo
de recomendação é feito utilizando avaliações com notas dos
usuários do movielens
          if (userTotalOfLikedAndRatedMovies > 0) {
               UserBasedRecommenderWithUserRatedData
userBasedRecommenderWithUserRatedData = new
UserBasedRecommenderWithUserRatedData(
```

```
                         new FileDataModel(new
File("data/movielens/movielens_ratings_rated_data.csv")));
               List<RecommendedItem>
recommendedItemsFromRecommenderWithRatedValues =
userBasedRecommenderWithUserRatedData

     .recommend(getPreferenceArrayWithRatedDataFromUser(),

     numberOfRecommendedItemsForRecommenderWithRatedData,
                                    getMultipleRescorer(1,
allRecommendedItems));

     allRecommendedItems.addAll(recommendedItemsFromRecommende
rWithRatedValues);
          }

          // Se houverem filmes assistidos, o cálculo de
recomendação é feito utilizando dados booleanos dos usuários
do movielens
          if (userTotalOfWatchedMovies > 0) {
               DataModel model = new
GenericBooleanPrefDataModel(GenericBooleanPrefDataModel
                         .toDataMap(new FileDataModel(new
File("data/movielens/movielens_ratings_boolean_data.csv"))));
               UserBasedRecommenderWithUserBooleanData
userBasedRecommenderWithUserBooleanData = new
UserBasedRecommenderWithUserBooleanData(
                         model);
               List<RecommendedItem>
recommendationsWithBooleanData =
userBasedRecommenderWithUserBooleanData.recommend(

     getPreferenceArrayWithBooleanDataFromUser(),
numberOfRecommendedItemsForRecommenderWithBooleanData,
                         getMultipleRescorer(2,
allRecommendedItems));

     allRecommendedItems.addAll(recommendationsWithBooleanData
);
          }

          // Retorna os resultados dos dois métodos de
avaliações, caso os dois tenham sido utilizados
          return allRecommendedItems;
     }

     public List<Movie>
getMovieRecommendations(List<RecommendedItem>
recommendedItems) {
          // Retorna os filmes recomendados
```

```
          List<Movie> recommendedMovies = new
ArrayList<Movie>();
          for (RecommendedItem recommendedItem :
recommendedItems) {

     recommendedMovies.add(MovieCRUD.getMovieById(recommendedI
tem.getItemID()));
          }
          return recommendedMovies;
     }

     public PreferenceArray
getPreferenceArrayWithRatedDataFromUser() {
          // Retorna os filmes curtidos/avaliados/assistidos
pelo usuário, com notas
          PreferenceArray userPreferences = new
GenericUserPreferenceArray(user.liked_rated_movies.size());
          userPreferences.setUserID(0,
PlusAnonymousUserDataModel.TEMP_USER_ID);
          int count = 0;
          for (UserLikedRatedMovie userLikeMovie :
user.liked_rated_movies) {
               userPreferences.setItemID(count,
userLikeMovie.movie.id);
               userPreferences.setValue(count,
userLikeMovie.rating);
               count++;
          }
          return userPreferences;
     }

     private PreferenceArray
getPreferenceArrayWithBooleanDataFromUser() {
          // Retorna os filmes curtidos/avaliados/assistidos
pelo usuário, sem notas
          PreferenceArray userPreferences = new
GenericUserPreferenceArray(user.watched_movies.size());
          userPreferences.setUserID(0,
PlusAnonymousUserDataModel.TEMP_USER_ID);
          int count = 0;
          for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {
               userPreferences.setItemID(count,
userWatchedMovie.movie.id);
               count++;
          }
          return userPreferences;
     }

     private ActorRescorer getActorRescorer() {
```

```
          return new
ActorRescorer(UserCRUD.getLikedActorsByUser(user));
     }

     private CountryRescorer getCountryRescorer() {
          return new CountryRescorer(

     UserMovieCountryCRUD.getTopCountriesWithRangeByUserFacebo
okId(user.id_user_facebook, 3));
     }

     private DirectorRescorer getDirectorRescorer() {
          return new
DirectorRescorer(UserCRUD.getLikedDirectorsByUser(user));
     }

     private GenreRescorer getGenreRescorer() {
          return new
GenreRescorer(UserMovieGenreCRUD.getTopGenresWithRangeByUserFa
cebookId(user.id_user_facebook, 4));
     }

     private MovieRatingRescorer getMovieRatingRescorer() {
          List<Float> userMoviesRatingsImdb = new
ArrayList<Float>();
          if (user.liked_rated_movies.size() > 0) {
               for (UserLikedRatedMovie userLikedRatedMovie :
user.liked_rated_movies) {

     userMoviesRatingsImdb.add(userLikedRatedMovie.movie.ratin
g_imdb);
               }
          }
          if (user.watched_movies.size() > 0) {
               for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {

     userMoviesRatingsImdb.add(userWatchedMovie.movie.rating_i
mdb);
               }
          }
          return new
MovieRatingRescorer(Collections.min(userMoviesRatingsImdb));
     }

     private MovieSimilarRescorer getMovieSimilarRescorer() {
          Set<Movie> userMoviesSimilar = new HashSet<Movie>();
          if (user.liked_rated_movies.size() > 0) {
               for (UserLikedRatedMovie userLikedRatedMovie :
user.liked_rated_movies) {
```

```
     userMoviesSimilar.addAll(MovieCRUD.getSimilarMoviesByMovi
e(userLikedRatedMovie.movie));
                }
           }
           if (user.watched_movies.size() > 0) {
                for (UserWatchedMovie userWatchedMovie :
user.watched_movies) {

     userMoviesSimilar.addAll(MovieCRUD.getSimilarMoviesByMovi
e(userWatchedMovie.movie));
                }
           }
           return new MovieSimilarRescorer(userMoviesSimilar);
     }

     public MultipleRescorer getMultipleRescorer(int
typeOfRecommender, List<RecommendedItem>
moviesAlreadyRecommended) {
           // Adiciona os rescorers
           List<IDRescorer> rescores = new
ArrayList<IDRescorer>();
           if (user.liked_rated_movies.size() > 0 ||
user.watched_movies.size() > 0) {
                if (typeOfRecommender == 1) {
                     if (user.liked_rated_movies.size() > 0) {

     rescores.add(getMovieSimilarRescorer());
                          if (user.liked_rated_movies.size() >
4) {

     rescores.add(getGenreRescorer());
                               if
(user.liked_rated_movies.size() > 9) {

     rescores.add(getCountryRescorer());

     rescores.add(getMovieRatingRescorer());
                               }
                          }
                     }
                } else {
                     if (user.watched_movies.size() > 0) {

     rescores.add(getMovieSimilarRescorer());
                          if (user.watched_movies.size() > 4) {

     rescores.add(getGenreRescorer());
                               if (user.watched_movies.size() >
9) {
```

```
     rescores.add(getCountryRescorer());

     rescores.add(getMovieRatingRescorer());
                              }
                         }
                    }
               }
               if (user.liked_actors.size() > 0) {
                    rescores.add(getActorRescorer());
               }
               if (user.liked_directors.size() > 0) {
                    rescores.add(getDirectorRescorer());
               }
          }
          MultipleRescorer multipleRescorer = new
MultipleRescorer(rescores, user, moviesAlreadyRecommended);
          // Retorna todos rescorers que foram adicionados
          return multipleRescorer;
     }
}
```

MovieMe/app/recommender/MovieSimilarRescorer.java

```
package recommender;

import java.util.Set;

import org.apache.mahout.cf.taste.recommender.IDRescorer;

import models.Movie;

public class MovieSimilarRescorer implements IDRescorer {

     private final Set<Movie>
similarMoviesToLikedRatedWatchedMovies;

     public MovieSimilarRescorer(Set<Movie>
similarMoviesToLikedMovies) {
          // Lista com os filmes similares aos filmes
curtidos/assistidos/avaliados pelo usuário
          this.similarMoviesToLikedRatedWatchedMovies =
similarMoviesToLikedMovies;
     }

     @Override
     public double rescore(long currentMovieId, double
originalScore) {
```

```
          // Aumenta o peso em 20% dos filmes que são
considerados similares aos filmes
curtidos/assistidos/avaliados pelo usuário
          for (Movie movie :
similarMoviesToLikedRatedWatchedMovies) {
               if (movie != null) {
                    if (movie.id == currentMovieId) {
                         return originalScore * 1.2;
                    }
               }
          }
          return originalScore;
     }

     @Override
     public boolean isFiltered(long currentMovieId) {
          return false;
     }

}
```

MovieMe/app/recommender/MultipleRescorer.java

```
package recommender;

import java.util.List;

import org.apache.mahout.cf.taste.recommender.IDRescorer;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;

import com.google.common.base.Preconditions;

import crud.MovieCRUD;
import crud.UserCRUD;
import crud.UserLikedRatedMovieCRUD;
import crud.UserWatchedMovieCRUD;
import models.Movie;
import models.User;

public final class MultipleRescorer implements IDRescorer {

     private final IDRescorer[] rescorers;
     private final User user;
     private final List<RecommendedItem>
moviesAlreadyRecommended;

     public MultipleRescorer(List<IDRescorer> rescorers, User
user, List<RecommendedItem> moviesAlreadyRecommended) {
          Preconditions.checkNotNull(rescorers);
```

```
          Preconditions.checkState(!rescorers.isEmpty(),
"rescorers estão vazios.");
          this.rescorers = rescorers.toArray(new
IDRescorer[rescorers.size()]);
          this.user = user;
          this.moviesAlreadyRecommended =
moviesAlreadyRecommended;
     }

     @Override
     public double rescore(long currentMovieId, double
originalScore) {
          // Adiciona todos os rescorers
          for (IDRescorer rescorer : rescorers) {
               if (rescorer != null) {
                    originalScore =
rescorer.rescore(currentMovieId, originalScore);
                    if (Double.isNaN(originalScore)) {
                         return Double.NaN;
                    }
               }
          }
          return originalScore;
     }

     @Override
     public boolean isFiltered(long currentMovieId) {
          // Filtra filmes de serem recomendados
          Movie currentMovie =
MovieCRUD.getMovieById(currentMovieId);
          if (currentMovie != null) {

               // Filtra filmes que já foram
curtidos/assistidos/avaliados pelo usuário
               if
(UserLikedRatedMovieCRUD.hasMovie(user.id_user_facebook,
currentMovie.id)
                         ||
UserWatchedMovieCRUD.hasMovie(user.id_user_facebook,
currentMovie.id)) {
                    return true;
               } else {

                    // Filtra filmes que já foram recomendados
pelo outro cálculo de recomendação
                    if (moviesAlreadyRecommended != null ||
!moviesAlreadyRecommended.isEmpty()) {
                         for (RecommendedItem recommendedItem
: moviesAlreadyRecommended) {
                              if (currentMovie.id ==
recommendedItem.getItemID()) {
```

```
                                    return true;
                               }
                          }
                     }

                     // Verifica se o filme é franquia
                     if (currentMovie.is_movie_franchise == 1)
{
                          // Verifica quais são os filmes
anteriores e posteriores da franquia e se o usuário já os
curtiu/assistiu/avaliou
                          // Necessário para filtrar filmes que
são de franquias e o usuário já tenha assistido algum da
franquia
                          // Por exemplo: se o usuário já
assistiu ao filme 3 da franquia, deve-se filtrar os filmes 1 e
2 da franquia
                          if (currentMovie.order_franchise !=
1) {
                               Movie previousFranchiseMovie =
MovieCRUD.getPreviousFranchiseMovieByMovieId(currentMovie.id);
                               if (previousFranchiseMovie !=
null) {
                                    if
(UserCRUD.movieWasLikedOrRatedByUser(user,
previousFranchiseMovie)
                                                        ||
UserCRUD.movieWasWatchedByUser(user, previousFranchiseMovie))
{
                                         for (Movie
nextFranchiseMovie : MovieCRUD

     .getNextFranchiseMoviesByMovieId(currentMovie.id)) {
                                              if
(UserCRUD.movieWasLikedOrRatedByUser(user, nextFranchiseMovie)
                                                             ||
UserCRUD.movieWasWatchedByUser(user, nextFranchiseMovie)) {
                                                   return true;
                                              }
                                         }
                                         return false;
                                    }
                                    return true;
                               }
                               return true;
                          } else {
                               for (Movie nextFranchiseMovie :
MovieCRUD.getNextFranchiseMoviesByMovieId(currentMovie.id)) {
                                    if
(UserCRUD.movieWasLikedOrRatedByUser(user, nextFranchiseMovie)
```

```
                                                    ||
UserCRUD.movieWasWatchedByUser(user, nextFranchiseMovie)) {
                                            return true;
                                        }
                                    }
                                    return false;
                                }
                            }
                            return false;
                        }
                    }
                    return true;
                }
```

MovieMe/app/recommender/UserBasedRecommenderWithBooleanData.java

```
package recommender;

import java.io.IOException;
import java.util.Collection;
import java.util.List;

import org.apache.mahout.cf.taste.common.Refreshable;
import org.apache.mahout.cf.taste.common.TasteException;
import
org.apache.mahout.cf.taste.impl.neighborhood.ThresholdUserNeig
hborhood;
import
org.apache.mahout.cf.taste.impl.recommender.GenericBooleanPref
UserBasedRecommender;
import
org.apache.mahout.cf.taste.impl.similarity.LogLikelihoodSimila
rity;
import org.apache.mahout.cf.taste.model.DataModel;
import
org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.IDRescorer;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.cf.taste.similarity.UserSimilarity;

public class UserBasedRecommenderWithBooleanData implements
Recommender {

    private final Recommender delegate;

    public UserBasedRecommenderWithBooleanData(DataModel
model) throws TasteException, IOException {
```

```
          // Recomendação utilizando dados com avaliações sem
notas - booleanas
          // Foram escolhidas a métrica de similridade e
abordagem de definição da vizinhança
          // que obteram melhores resultados na avaliação off-
line para esse contexto
          UserSimilarity similarity = new
LogLikelihoodSimilarity(model);
          UserNeighborhood neighborhood = new
ThresholdUserNeighborhood(0.7, similarity, model);
          delegate = new
GenericBooleanPrefUserBasedRecommender(model, neighborhood,
similarity);
     }

     @Override
     public void refresh(Collection<Refreshable>
alreadyRefreshed) {
          delegate.refresh(alreadyRefreshed);
     }

     @Override
     public DataModel getDataModel() {
          return delegate.getDataModel();
     }

     @Override
     public List<RecommendedItem> recommend(long userID, int
howMany) throws TasteException {
          return delegate.recommend(userID, howMany);
     }

     @Override
     public List<RecommendedItem> recommend(long userID, int
howMany, IDRescorer rescorer) throws TasteException {
          return delegate.recommend(userID, howMany,
rescorer);
     }

     @Override
     public void removePreference(long userID, long itemID)
throws TasteException {
          delegate.removePreference(userID, itemID);
     }

     @Override
     public void setPreference(long userID, long itemID, float
value) throws TasteException {
          delegate.setPreference(userID, itemID, value);
     }
```

```
     @Override
     public float estimatePreference(long userID, long itemID)
throws TasteException {
          return delegate.estimatePreference(userID, itemID);
     }

}
```

MovieMe/app/recommender/UserBasedRecommenderWithRatedData.java

```
package recommender;

import java.io.IOException;
import java.util.Collection;
import java.util.List;

import org.apache.mahout.cf.taste.common.Refreshable;
import org.apache.mahout.cf.taste.common.TasteException;
import
org.apache.mahout.cf.taste.impl.neighborhood.NearestNUserNeigh
borhood;
import
org.apache.mahout.cf.taste.impl.recommender.GenericUserBasedRe
commender;
import
org.apache.mahout.cf.taste.impl.similarity.EuclideanDistanceSi
milarity;
import org.apache.mahout.cf.taste.model.DataModel;
import
org.apache.mahout.cf.taste.neighborhood.UserNeighborhood;
import org.apache.mahout.cf.taste.recommender.IDRescorer;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;
import org.apache.mahout.cf.taste.recommender.Recommender;
import org.apache.mahout.cf.taste.similarity.UserSimilarity;

public class UserBasedRecommenderWithRatedData implements
Recommender {

     private final Recommender delegate;

     public UserBasedRecommenderWithRatedData(DataModel model)
throws TasteException, IOException {
          // Recomendação utilizando dados com avaliações com
notas
          // Foram escolhidas a métrica de similridade e
abordagem de definição da vizinhança
          // que obteram melhores resultados na avaliação off-
line para esse contexto
          UserSimilarity similarity = new
EuclideanDistanceSimilarity(model);
```

```
          UserNeighborhood neighborhood = new
NearestNUserNeighborhood(200, similarity, model);
          delegate = new GenericUserBasedRecommender(model,
neighborhood, similarity);
     }

     @Override
     public void refresh(Collection<Refreshable>
alreadyRefreshed) {
          delegate.refresh(alreadyRefreshed);
     }

     @Override
     public DataModel getDataModel() {
          return delegate.getDataModel();
     }

     @Override
     public List<RecommendedItem> recommend(long userID, int
howMany) throws TasteException {
          return delegate.recommend(userID, howMany);
     }

     @Override
     public List<RecommendedItem> recommend(long userID, int
howMany, IDRescorer rescorer) throws TasteException {
          return delegate.recommend(userID, howMany,
rescorer);
     }

     @Override
     public void removePreference(long userID, long itemID)
throws TasteException {
          delegate.removePreference(userID, itemID);
     }

     @Override
     public void setPreference(long userID, long itemID, float
value) throws TasteException {
          delegate.setPreference(userID, itemID, value);
     }

     @Override
     public float estimatePreference(long userID, long itemID)
throws TasteException {
          return delegate.estimatePreference(userID, itemID);
     }

}
```

MovieMe/app/recommender/UserBasedRecommenderWithUserBooleanData.java

```
package recommender;

import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import
org.apache.mahout.cf.taste.impl.model.PlusAnonymousUserDataMod
el;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.model.PreferenceArray;
import org.apache.mahout.cf.taste.recommender.IDRescorer;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;

public class UserBasedRecommenderWithUserBooleanData extends
UserBasedRecommenderWithBooleanData {

     private final PlusAnonymousUserDataModel
plusAnonymousModel;

     public UserBasedRecommenderWithUserBooleanData(DataModel
model) throws TasteException, IOException {
          super(new PlusAnonymousUserDataModel(model));
          plusAnonymousModel = (PlusAnonymousUserDataModel)
getDataModel();
     }

     public synchronized List<RecommendedItem>
recommend(PreferenceArray anonymousUserPrefs, int howMany,
               IDRescorer rescorer) throws TasteException {
          // Adiciona ao cálculo de recomendação os dados
advindos do Facebook do usuário
          plusAnonymousModel.setTempPrefs(anonymousUserPrefs);
          List<RecommendedItem> recommendations =
recommend(PlusAnonymousUserDataModel.TEMP_USER_ID, howMany,
rescorer);
          plusAnonymousModel.clearTempPrefs();
          return recommendations;
     }

}
```

MovieMe/app/recommender/UserBasedRecommenderWithUserRatedData.java

```
package recommender;
```

```
import java.io.IOException;
import java.util.List;

import org.apache.mahout.cf.taste.common.TasteException;
import
org.apache.mahout.cf.taste.impl.model.PlusAnonymousUserDataMod
el;
import org.apache.mahout.cf.taste.model.DataModel;
import org.apache.mahout.cf.taste.model.PreferenceArray;
import org.apache.mahout.cf.taste.recommender.IDRescorer;
import org.apache.mahout.cf.taste.recommender.RecommendedItem;

public class UserBasedRecommenderWithUserRatedData extends
UserBasedRecommenderWithRatedData {

     private final PlusAnonymousUserDataModel
plusAnonymousModel;

     public UserBasedRecommenderWithUserRatedData(DataModel
model) throws TasteException, IOException {
          super(new PlusAnonymousUserDataModel(model));
          plusAnonymousModel = (PlusAnonymousUserDataModel)
getDataModel();
     }

     public synchronized List<RecommendedItem>
recommend(PreferenceArray anonymousUserPrefs, int howMany,
               IDRescorer rescorer) throws TasteException {
          // Adiciona ao cálculo de recomendação os dados
advindos do Facebook do usuário
          plusAnonymousModel.setTempPrefs(anonymousUserPrefs);
          List<RecommendedItem> recommendations =
recommend(PlusAnonymousUserDataModel.TEMP_USER_ID, howMany,
rescorer);
          plusAnonymousModel.clearTempPrefs();
          return recommendations;
     }

}
```

package services

MovieMe/app/services/FacebookService.java

```
package services;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
```

```
import java.util.regex.Pattern;

import org.pac4j.oauth.profile.facebook.FacebookProfile;

import com.restfb.Connection;
import com.restfb.DefaultFacebookClient;
import com.restfb.FacebookClient;
import com.restfb.Parameter;
import com.restfb.Version;
import com.restfb.json.JsonObject;
import com.restfb.types.Page;

public class FacebookService {

     private String facebookUserId;
     private FacebookClient facebookClient;

     public FacebookService(FacebookProfile facebookProfile) {
          facebookUserId = facebookProfile.getId();
          facebookClient = new
DefaultFacebookClient(facebookProfile.getAccessToken(),
Version.LATEST);
     }

     public List<Page> getLikedMovies() {
          // Retorna páginas de filmes curtidas pelo usuário
          Connection<Page> likedFacebookMovies =
facebookClient.fetchConnection(facebookUserId + "/movies",
Page.class,
                    Parameter.with("limit", 100),
Parameter.with("fields", "id, name, release_date,
directed_by"));

          List<Page> likedMovies = new ArrayList<Page>();

          likedMovies.addAll(likedFacebookMovies.getData());

          while (likedFacebookMovies.hasNext()) {
               likedFacebookMovies =
facebookClient.fetchConnectionPage(likedFacebookMovies.getNext
PageUrl(), Page.class);

     likedMovies.addAll(likedFacebookMovies.getData());
          }

          return likedMovies;
     }

     public List<Page> getLikedActorsAndDirectors() {
          // Retorna páginas de atores e/ou diretores curtidas
pelo usuário
```

```
          Connection<Page> facebookLikes =
facebookClient.fetchConnection(facebookUserId + "/likes",
Page.class,
                    Parameter.with("fields", "name,
category"), Parameter.with("limit", 100));

          List<Page> likedActorsDirectors = new
ArrayList<Page>();

          for (Page page : facebookLikes.getData()) {
               if (page.getCategory().equals("Actor/Director")
|| page.getCategory().equals("Public Figure")
                         ||
page.getCategory().equals("Artist")) {
                    likedActorsDirectors.add(page);
               }
          }

          while (facebookLikes.hasNext()) {
               facebookLikes =
facebookClient.fetchConnectionPage(facebookLikes.getNextPageUr
l(), Page.class);
               for (Page page : facebookLikes.getData()) {
                    if
(page.getCategory().equals("Actor/Director") ||
page.getCategory().equals("Public Figure")
                              ||
page.getCategory().equals("Artist")) {
                         likedActorsDirectors.add(page);
                    }
               }
          }

          return likedActorsDirectors;
     }

     public List<Page> getLikes() {
          // Retorna páginas curtidas pelo usuário (apenas as
relacionadas ao contexto)
          Connection<Page> facebookLikes =
facebookClient.fetchConnection(facebookUserId + "/likes",
Page.class,
                    Parameter.with("fields", "id, name,
category"), Parameter.with("limit", 100));

          List<Page> allLikes = new ArrayList<Page>();

          for (Page page : facebookLikes.getData()) {
               if (page.getCategory().equals("Movie") ||
page.getCategory().equals("Actor/Director")
```

```
                         || page.getCategory().equals("Public
Figure") || page.getCategory().equals("Fictional Character")
                         ||
page.getCategory().equals("Artist")) {
                    allLikes.add(page);
               }
          }

          while (facebookLikes.hasNext()) {
               facebookLikes =
facebookClient.fetchConnectionPage(facebookLikes.getNextPageUr
l(), Page.class);
               for (Page page : facebookLikes.getData()) {
                    if (page.getCategory().equals("Movie") ||
page.getCategory().equals("Actor/Director")
                              ||
page.getCategory().equals("Public Figure")
                              ||
page.getCategory().equals("Fictional Character") ||
page.getCategory().equals("Artist")) {
                         allLikes.add(page);
                    }
               }
          }

          return allLikes;
     }

     public List<Page> getWatchedMovies() {
          // Retorna filmes marcados como assistidos pelo
usuário
          Connection<JsonObject> jsonObjectWatchedMovies =
facebookClient
                    .fetchConnection(facebookUserId +
"/video.watches", JsonObject.class, Parameter.with("limit",
100));

          List<Page> watchedMovies = new ArrayList<Page>();

          JsonObject jsonObjectData = null;
          JsonObject jsonObjectMovie = null;

          for (JsonObject jsonObjectWatchedMovie :
jsonObjectWatchedMovies.getData()) {
               if (jsonObjectWatchedMovie.has("data")) {
                    jsonObjectData =
jsonObjectWatchedMovie.getJsonObject("data");
                    if (jsonObjectData.has("movie")) {
                         jsonObjectMovie =
jsonObjectData.getJsonObject("movie");
```

```
                        if (jsonObjectMovie.has("id") &&
jsonObjectMovie.has("url")) {
                            if
(jsonObjectMovie.getString("url").toLowerCase().contains("face
book")) {

     watchedMovies.add(facebookClient.fetchObject("/" +
jsonObjectMovie.getString("id"),
                                                Page.class,
Parameter.with("fields", "id, name, release_date,
directed_by")));
                            }
                        }
                    }
                }
           }

           while (jsonObjectWatchedMovies.hasNext()) {
                jsonObjectWatchedMovies =
facebookClient.fetchConnectionPage(jsonObjectWatchedMovies.get
NextPageUrl(),
                        JsonObject.class);
                for (JsonObject jsonObjectWatchedMovie :
jsonObjectWatchedMovies.getData()) {
                    if (jsonObjectWatchedMovie.has("data")) {
                        jsonObjectData =
jsonObjectWatchedMovie.getJsonObject("data");
                        if (jsonObjectData.has("movie")) {
                            jsonObjectMovie =
jsonObjectData.getJsonObject("movie");
                            if (jsonObjectMovie.has("id") &&
jsonObjectMovie.has("url")) {
                                if
(jsonObjectMovie.getString("url").toLowerCase().contains("face
book")) {

     watchedMovies.add(facebookClient.fetchObject("/" +
jsonObjectMovie.getString("id"),
                                                    Page.class,
Parameter.with("fields", "id, name, release_date,
directed_by")));
                                }
                            }
                        }
                    }
                }
           }

           return watchedMovies;
     }
```

```
     public Map<Page, Integer> getRatedMovies() {
          // Retorna filmes avaliados pelo usuário
          Connection<JsonObject> jsonObjectRatedMovies =
facebookClient.fetchConnection(facebookUserId +
"/video.rates",
                    JsonObject.class, Parameter.with("limit",
100));

          Map<Page, Integer> ratedMoviesPageAndScore = new
HashMap<Page, Integer>();

          JsonObject jsonObjectData = null;
          JsonObject jsonObjectRating = null;
          JsonObject jsonObjectMovie = null;

          for (JsonObject jsonObjectRatedMovie :
jsonObjectRatedMovies.getData()) {
               if (jsonObjectRatedMovie.has("data")) {
                    jsonObjectData =
jsonObjectRatedMovie.getJsonObject("data");
                    if (jsonObjectData.has("rating") &&
jsonObjectData.has("movie")) {
                         jsonObjectRating =
jsonObjectData.getJsonObject("rating");
                         jsonObjectMovie =
jsonObjectData.getJsonObject("movie");
                         if (jsonObjectRating.has("value") &&
jsonObjectMovie.has("id")) {
                              if
(jsonObjectMovie.getString("url").toLowerCase().contains("face
book")) {

     ratedMoviesPageAndScore.put(

     facebookClient.fetchObject("/" +
jsonObjectMovie.getString("id"), Page.class,

     Parameter.with("fields", "id, name, release_date,
directed_by")),

     jsonObjectRating.getInt("value"));
                              }
                         }
                    }
               }
          }

          while (jsonObjectRatedMovies.hasNext()) {
               jsonObjectRatedMovies =
facebookClient.fetchConnectionPage(jsonObjectRatedMovies.getNe
xtPageUrl(),
```

```
                        JsonObject.class);
                for (JsonObject jsonObjectRatedMovie :
jsonObjectRatedMovies.getData()) {
                    if (jsonObjectRatedMovie.has("data")) {
                        jsonObjectData =
jsonObjectRatedMovie.getJsonObject("data");
                        if (jsonObjectData.has("rating") &&
jsonObjectData.has("movie")) {
                            jsonObjectRating =
jsonObjectData.getJsonObject("rating");
                            jsonObjectMovie =
jsonObjectData.getJsonObject("movie");
                            if
(jsonObjectRating.has("value") && jsonObjectMovie.has("id")) {
                                if
(jsonObjectMovie.getString("url").toLowerCase().contains("face
book")) {

    ratedMoviesPageAndScore.put(

    facebookClient.fetchObject("/" +
jsonObjectMovie.getString("id"), Page.class,

    Parameter.with("fields", "id, name, release_date,
directed_by")),

    jsonObjectRating.getInt("value"));
                                }
                            }
                        }
                    }
            }

            return ratedMoviesPageAndScore;
    }

    public String normalizeFacebookMovieTitle(String
movieTitle) {
            // Normaliza o título da página do filme
            Pattern ignoreWords = Pattern.compile(
                    "\\b(?: filme| filmes| o filme| - filme| -
filmes| - o filme| movie| movies| the movie| movie page|
movies page| - movie| - movies| - the movie| film| film
series| films| the film| - film| - films| - the film|
trilogia| a trilogia| - trilogia| trilogy| - trilogy| brasil|
br| - brasil| brazil| - brazil| - br| franquia| - franquia|
the imax experience| 3d| in 3d| oficial| - oficial| official|
- official)\\b\\s*",
                    Pattern.CASE_INSENSITIVE);
```

```
          Matcher matcher =
ignoreWords.matcher(movieTitle.replace(":", " -").replace(" e
", " & ")
                    .replace(" and ", " & ").replaceAll("
\\((.+?)\\)", "").replace("\u2122", ""));
          return matcher.replaceAll("");
     }

     public String normalizeFacebookActorDirectorName(String
actorDirectorName) {
          // Normaliza o título da página do ator e/ou diretor
          Pattern ignoreWords = Pattern.compile(
                    "\\b(?: brasil| - brasil| brazil| -
brazil| br| - br| lovers| - lovers| fans| - fans| society| -
society| news| - news| oficial| - oficial| official| -
official| official page| - official page)\\b\\s*",
                    Pattern.CASE_INSENSITIVE);
          Matcher matcher =
ignoreWords.matcher(actorDirectorName.replaceAll("
\\((.+?)\\)", ""));
          return matcher.replaceAll("");
     }

     public void setMoviePageReleaseData(Page moviePage) {
          // Identifica o ano de lançamento em diferentes
formatos de data presentes no atributo data de lançamento, se
houver
          // Tenta identificar o ano de lançamento no título
da página do filme, caso não esteja presente o atributo data
de lançamento
          if (moviePage.getReleaseDate() == null) {
               Pattern yearInsideParentheses =
Pattern.compile("\\((\\d{4})\\)*");
               Matcher matcher =
yearInsideParentheses.matcher(moviePage.getName());
               if (matcher.find()) {

     moviePage.setReleaseDate(matcher.group(1));
               }
          } else {
               Pattern yearInsideReleaseDate =
Pattern.compile("(\\d{4})");
               Matcher matcher =
yearInsideReleaseDate.matcher(moviePage.getReleaseDate());
               if (matcher.find()) {

     moviePage.setReleaseDate(matcher.group(1));
               } else {
                    moviePage.setReleaseDate(null);
               }
          }
```

```
     }
}
```

MovieMe/app/services/MyApiFilmsJsonService.java

```
package services;

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Iterator;

import org.apache.commons.io.IOUtils;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

import models.Movie;

public class MyApiFilmsJsonService {

     // URL do MyAPIFilmes para consultar base de dados do
IMDB pelo título do filme
     private static final String URL_IMDB_FIND_MOVIE_BY_TITLE
=
"http://www.myapifilms.com/imdb/idIMDB?title=MOVIE_TITLE&token
=a3dbcefc-aecf-40d7-8d2f-de5cfda2e037&format=json&language=pt-
br&filter=3&limit=1&&actors=1&aka=1&similarMovies=1&fullSize=1
";
     // URL do MyAPIFilmes para consultar base de dados do
IMDB pelo id do IMDB do filme
     private static final String
URL_IMDB_FIND_MOVIE_BY_ID_IMDB =
"http://www.myapifilms.com/imdb/idIMDB?idIMDB=MOVIE_ID_IMDB&to
ken=a3dbcefc-aecf-40d7-8d2f-
de5cfda2e037&format=json&language=pt-
br&filter=3&limit=1&&actors=1&aka=1&similarMovies=1&fullSize=1
";
     // URL do MyAPIFilmes para consultar base de dados do
TMDB pelo id do IMDB do filme
     private static final String
URL_TMDB_FIND_MOVIE_BY_ID_IMDB =
"http://www.myapifilms.com/tmdb/movieInfoImdb?idIMDB=MOVIE_ID_
IMDB&token=a3dbcefc-aecf-40d7-8d2f-
de5cfda2e037&format=json&language=pt";

     public MyApiFilmsJsonService() {
```

```
     }

     public static String getIdImdbByMovieTitle(String
movieTitle)
               throws JsonProcessingException,
MalformedURLException, IOException {
          String urlImdbFindMovieByTitle =
URL_IMDB_FIND_MOVIE_BY_TITLE.replace("MOVIE_TITLE",
movieTitle);

          Iterator<JsonNode> jsonNodesMovies =
getJsonNodeFromUrl(urlImdbFindMovieByTitle, 1).iterator();

          return
jsonNodesMovies.next().path("idIMDB").asText();
     }

     public static Movie getMovieByMovieTitle(String
movieTitle)
               throws JsonProcessingException,
MalformedURLException, IOException {
          String urlImdbFindMovieByTitle =
URL_IMDB_FIND_MOVIE_BY_TITLE.replace("MOVIE_TITLE",
movieTitle);

          ObjectMapper objectMapper = new ObjectMapper();

          // Ignora atributos não identificados

     objectMapper.configure(DeserializationFeature.FAIL_ON_UNK
NOWN_PROPERTIES, false);

          Movie[] movieFromImdb =
objectMapper.treeToValue(getJsonNodeFromUrl(urlImdbFindMovieBy
Title, 1), Movie[].class);

          // Quando o título em português é o mesmo que o
título original, o campo do título original vem vazio
          if (movieFromImdb[0].title_original == null ||
movieFromImdb[0].title_original.equals("")) {
               movieFromImdb[0].title_original =
movieFromImdb[0].title_br;
          }

          return movieFromImdb[0];
     }

     public static Movie getMovieByIdImdb(String idImdb)
               throws JsonProcessingException,
MalformedURLException, IOException {
          String idImdbFormatted = "tt" + idImdb;
```

```
          String urlImdbFindMovieByIdImdb =
URL_IMDB_FIND_MOVIE_BY_ID_IMDB.replace("MOVIE_ID_IMDB",
idImdbFormatted);
          // Necessário para obter a sinopse em português -
IMDB só disponibiliza em inglês
          // Necessário para obter a URL da imagem do poster -
IMDB bloqueia suas URLs de posters para uso externo
          String urlTmdbFindMovieByIdImdb =
URL_TMDB_FIND_MOVIE_BY_ID_IMDB.replace("MOVIE_ID_IMDB",
idImdbFormatted);

          ObjectMapper objectMapper = new ObjectMapper();

          // Ignora atributos não identificados

     objectMapper.configure(DeserializationFeature.FAIL_ON_UNK
NOWN_PROPERTIES, false);

          Movie[] movieFromImdb =
objectMapper.treeToValue(getJsonNodeFromUrl(urlImdbFindMovieBy
IdImdb, 1),
                    Movie[].class);

          // Quando o título em português é o mesmo que o
título original, o campo do título original vem vazio
          if (movieFromImdb[0].title_original == null ||
movieFromImdb[0].title_original.equals("")) {
               movieFromImdb[0].title_original =
movieFromImdb[0].title_br;
          }

          JsonNode movieFromTmdbPlotBr =
getJsonNodeFromUrl(urlTmdbFindMovieByIdImdb, 2);
          JsonNode movieFromTmdbUrlPoster =
getJsonNodeFromUrl(urlTmdbFindMovieByIdImdb, 2);

          // Sinopse em português e URL do poster do filme
          if (movieFromTmdbPlotBr != null) {
               movieFromImdb[0].plot_br =
movieFromTmdbPlotBr.textValue();
               movieFromImdb[0].url_poster =
movieFromTmdbUrlPoster.textValue();
          }

          return movieFromImdb[0];
     }

     private static JsonNode getJsonNodeFromUrl(String url,
int imdbOrTmdbPlotBrOrTmdbUrlPoster)
               throws JsonProcessingException,
MalformedURLException, IOException {
```

```
          ObjectMapper objectMapper = new ObjectMapper();

          JsonNode jsonNode =
objectMapper.readTree(getJsonFromUrl(url));

          // (1) URL IMDB
          if (imdbOrTmdbPlotBrOrTmdbUrlPoster == 1) {
               return jsonNode.get("data").get("movies");
          } else {
               if (jsonNode.has("data")) {
                    // (2) URL TMDB - Sinopse em português
                    if (imdbOrTmdbPlotBrOrTmdbUrlPoster == 2)
{
                         if
(jsonNode.get("data").has("overview")) {
                              return
jsonNode.get("data").get("overview");
                         } else {
                              return null;
                         }
                         // (3) URL TMDB - URL da imagem do
poster
                    } else if (imdbOrTmdbPlotBrOrTmdbUrlPoster
== 3) {
                         if
(jsonNode.get("data").has("poster_path")) {
                              return
jsonNode.get("data").get("poster_path");
                         } else {
                              return null;
                         }
                    } else {
                         return null;
                    }
               } else {
                    return null;
               }
          }
     }

     private static String getJsonFromUrl(String url) throws
MalformedURLException, IOException {
          return IOUtils.toString(new URL(url));
     }

}
```

package views

MovieMe/app/views/about.scala.html

```
@main("Sobre") {
     <div class="wrapper">
         @headerPublic()
         <!-- Main content -->
         <section class="container">
             <div class="col-sm-12">
                 <div class="basic-container">
                     <div class="basic-container-
head">Sobre</div>
                     <div class="basic-container-describe">
                     Trabalho de Conclusão de Curso - Sistemas
de Informação / INE / CTC / UFSC<br/><br/>
                     Information courtesy of <a
href="http://www.imdb.com" target="_blank">IMDb</a>. Used with
permission.<br/><br/>
                     This product uses the TMDb API but is not
endorsed or certified by TMDb.<br/><br/>
                     <a href="https://www.themoviedb.org"
target="_blank"><img alt='logo'
src="@routes.Assets.at("images/logo-tmdb.png")"></a><br/><br/>
                     </div>
                     <div class="basic-container-content">
                         <a href="/" class="btn btn-md btn-
red">Voltar para página inicial</a>
                     </div>
                 </div>
             </div>
         </section>
          <div class="push"></div>
     </div>
     @footer()
}
```

MovieMe/app/views/admin.scala.html

```
@(isAdmin : Boolean, noData : Boolean)
@main("Painel Administrativo") {
     <div class="wrapper">
         @headerPublic()
         <!-- Main content -->
         <section class="container">
             <div class="col-sm-12">
                 <div class="basic-container">
                     <div class="basic-container-head">Painel
Administrativo</div>
                     <div class="basic-container-describe">
                         <br/><br/><br/>
```

```
                        @if(isAdmin) {
                                @if(noData) {
                                <a
href="@routes.AdminController.importMoviesAndRelated()"
class="btn btn-md btn-red">Importar filmes e dados
relacionados para a Base de Dados</a>
                                } else {
                                <div class="alert alert-
success">
                                     <span class='icon-
flag'></span>
                                          Filme e dados
relacionados já importados!
                                </div>
                                }
                        } else {
                                <div class="alert alert-error">
                                <span class='icon-
flag'></span>
                                     Apenas o usuário
Administrador tem permissão para acessar o Painel
Administrativo!
                                </div>
                        }
                    </div>
                </div>
            </div>
        </section>
         <div class="push"></div>
    </div>
    @footer()
}
```

MovieMe/app/views/error.scala.html

```
@main("Erro!") {
     <div class="wrapper">
          <div class="error">
               <h1 class="error-text">Ocorreu um erro!.</h1>
               <a href="/" class="btn btn-md btn-
yellow">Retornar para a página inicial</a>
          </div>
     </div>
}
```

MovieMe/app/views/error404.scala.html

```
@main("Página não Encontrada!") {
     <div class="wrapper">
          <div class="error">
```

```
                <img alt=""
src="@routes.Assets.at("images/error.png")" class="error-
image">
                <h1 class="error-text">Página não
Encontrada!</h1>
                <a href="/" class="btn btn-md btn-
yellow">Retornar para a página inicial</a>
           </div>
     </div>
}
```

MovieMe/app/views/evaluations.scala.html

```
@(isAdmin : Boolean, mae : String, rmse : String, precision :
String, recall : String)
@main("Avaliação Offline") {
     <div class="wrapper">
         @headerPublic()
         <!-- Main content -->
         <section class="container">
             <div class="col-sm-12">
                 <div class="basic-container">
                     <div class="basic-container-
head">Avaliação Offline</div>
                     <div class="basic-container-describe">
                     @if(isAdmin) {
                          <strong>MAE: </strong>@mae<br/><br/>
                          <strong>RMSE:
</strong>@rmse<br/><br/>
                          <strong>Precision:
</strong>@precision<br/><br/>
                          <strong>Recall:
</strong>@recall<br/><br/>
                      } else {
                         <br/><br/>
                         <div class="alert alert-error">
                              <span class='icon-flag'></span>
                                   Apenas o usuário
Administrador tem permissão para acessar essa página!
                         </div>
                      }
                     </div>
                 </div>
             </div>
         </section>
          <div class="push"></div>
     </div>
     @footer()
}
```

MovieMe/app/views/footer.scala.html

```
<footer class="footer-wrapper">
     <p class="copy">&copy; 2016 Movie.Me<br/>
         <a href="@routes.ContentController.about()">Sobre</a>
|
         <a href="#">Política de Privacidade</a> |
         <a href="#">Termos de Serviço</a>
     </p>
</footer>
<!-- JavaScript-->
<!-- jQuery 2.2.4 -->
<script src="@routes.Assets.at("js/jquery-
2.2.4.min.js")"></script>
<script src="@routes.Assets.at("js/spin.min.js")"></script>
<!-- Bootstrap -->
<script
src="@routes.Assets.at("js/bootstrap.min.js")"></script>
<script type="text/javascript">
function init_Elements(){$(".btn-user-
menu").click(function(e){e.preventDefault(),$(".user-
menu").toggleClass("user-menu-open")})}function
saveUserMovieRecommendationApproved(e,o,n){$.ajax({url:e,type:
"POST",success:function(){0==n?($("a#disapproved-
movie"+o).text("Recomendação Reprovada!"),$("a#disapproved-
movie"+o).addClass("disabled"),$("a#approved-
movie"+o).removeClass("disabled"),$("a#approved-
movie"+o).text("Aprovar Recomendação")):1==n&&($("a#approved-
movie"+o).text("Recomendação Aprovada!"),$("a#approved-
movie"+o).addClass("disabled"),$("a#disapproved-
movie"+o).removeClass("disabled"),$("a#disapproved-
movie"+o).text("Reprovar
Recomendação"))}})}$(document).ready(function(){init_Elements(
),$("#button-
recommender").on("click",function(){$(this).button("loading");
var e=document.getElementById("spinner");(new
Spinner).spin(e);$("#spinner").show()}),$("#login-
facebook").on("click",function(){$(this).button("loading")}),$
("#form-user-
opinion").submit(function(e){e.preventDefault();var
o=$("input[name=form-user-opinion-rate]:checked","#form-user-
opinion").val(),n=$("#user-message").val();"undefined"==typeof
o?($(".alert-success").hide(),$(".alert-
error").show()):jsRoutes.controllers.RecommenderController.sav
eUserRecommendationEvaluationOpinion(o,n).ajax({success:functi
on(){$("input[name=form-user-opinion-
rate]").prop("checked",!1),$("#user-
message").val("").blur(),$("#button-form-user-
opinion").unbind("click"),$(".alert-error").hide(),$(".alert-
success").show()}})})});
</script>
```

MovieMe/app/views/formUserOpinion.scala.html

```
@(totalUserLikedMovies: Integer, totalUserWatchedMovies:
Integer, totalUserRatedMovies : Integer,
totalUserLikedActorsDirectors: Integer)
<div class="form-user-opinion-wrapper">
     <div class="container">
          <div class="col-sm-12 col-md-10 col-md-offset-1 col-
lg-8 col-lg-offset-2">
               <!-- Alert succcess -->
               <div class="alert alert-success"
style="display: none;">
                    <span class='icon-flag'></span>Avaliação
realizada com sucesso!
               </div>
               <!-- Alert error -->
               <div class="alert alert-error" style="display:
none;">
                    <span class='icon-warning'></span>Por
favor, selecione uma avaliação!
               </div>
               <form id="form-user-opinion" class="form-user-
opinion">
                    <p class="form-user-opinion-tittle">Deixe
sua opinião</p>
                    <p class="form-user-opinion-subtittle">
                         Considerando que você:<br/>
                         Curtiu @totalUserLikedMovies filmes;
                         Assistiu: @totalUserWatchedMovies
filmes;
                         Avaliou: @totalUserRatedMovies
filmes;
                         Curtiu @totalUserLikedActorsDirectors
atores/diretores/relacionados...
                    </p>
                    <p class="form-user-opinion-question">Como
você avalia os filmes recomendados?</p>
                    <div class="col-sm-12">
                         <div class="radio-buttons">
                              <label class="radio-
inline"><input type="radio" name="form-user-opinion-rate"
value="1">Excelente</label>
                              <label class="radio-
inline"><input type="radio" name="form-user-opinion-rate"
value="2">Bom</label>
                              <label class="radio-
inline"><input type="radio" name="form-user-opinion-rate"
value="3">Regular</label>
```

```
                                <label class="radio-
inline"><input type="radio" name="form-user-opinion-rate"
value="4">Ruim</label>
                            </div>
                            <textarea placeholder="Explique sua
escolha (opcional)" id="user-message" name="user-message"
class="form-user-opinion-message"></textarea>
                        </div>
                        <button type="submit" id="button-form-
user-opinion" class='btn btn-md btn-red'>Enviar</button>
                    </form>
               </div>
          </div>
</div>
```

MovieMe/app/views/header.scala.html

```
@(userName : String, enoughData: Boolean, hasRecommendations :
Boolean)
<!-- Header section -->
<header class="header-wrapper">
     <div class="container">
          <!-- Logo -->
          <a href='/' class="logo">
               <img alt='logo'
src="@routes.Assets.at("images/logo.png")">
          </a>
          <!-- User menu-->
          <div class="control-panel">
               <a href="#" class="btn btn-user-
menu">@userName</a>
               <ul class="user-menu">
                    @if(enoughData && hasRecommendations) {
                         <li><a href="/movies" class="user-
menu-item">Filmes Recomendados</a></li>
                    }
                    <li><a href="/logout" class="user-menu-
item">Sair</a></li>
               </ul>
          </div>
     </div>
</header>
```

MovieMe/app/views/headerPublic.scala.html

```
<!-- Header section -->
<header class="header-wrapper">
     <div class="container">
          <!-- Logo -->
          <a href='/' class="logo">
```

```
                <img alt='logo'
src="@routes.Assets.at("images/logo.png")">
            </a>
        </div>
</header>
```

MovieMe/app/views/index.scala.html

```
@(userName : String, enoughData: Boolean, hasRecommendations :
Boolean)
@main("Início") {
     <div class="wrapper">
         @header(userName, enoughData, hasRecommendations)
         <!-- Main content -->
         <section class="container">
             <div class="col-sm-12">
                 <div class="basic-container">
                     <div class="basic-container-head">Olá,
@userName!</div>
                     <div class="basic-container-describe">
                          O <strong>Movie.Me</strong> se
baseia nos filmes que você curtiu, assistiu e/ou avaliou no
Facebook para
                          gerar as recomendações. Também
utiliza curtidas relacionadas a filmes, como páginas de
diretores e atores.
                          <br/><br/>
                          Clique no botão abaixo para gerar
recomendações de filmes baseado em seu perfil no Facebook.
                          <br>
                     </div>
                     <div class="basic-container-content">
                         @if(enoughData) {
                              <a href="/movies" class="btn
btn-md btn-red" id="button-recommender" data-loading-
text="Carregando...">
                                   Gerar recomendações
                              </a>
                              <div id="spinner"></div>
                         } else {
                              <br/>
                              <div class="alert alert-error">
                                   <span class='icon-
warning'></span>
                                        <strong>ERRO:</strong>
Dados insuficientes para gerar recomendações. Você não
curtiu/assistiu/avaliou
                                        nenhum filme no
Facebook!
                              </div>
```

```
                        }
                    </div>
                </div>
            </div>
        </section>
         <div class="push"></div>
    </div>
    @footer()
}
```

MovieMe/app/views/login.scala.html

```
@(urlFacebook: String)
@main("Login") {
    <div class="wrapper">
        <!-- Header section -->
        <header class="header-wrapper">
            <div class="container">
                <!-- Logo -->
                <a href='/' class="logo">
                    <img alt='logo'
src="@routes.Assets.at("images/logo.png")">
                </a>
            </div>
        </header>

        <!-- Main content -->
        <section class="container">
            <div class="col-sm-12">
                <div class="basic-container">
                    <div class="basic-container-
head">Entrar</div>
                    <div class="basic-container-
describe">Realize o login pelo Facebook</div>
                    <div class="basic-container-content">
                        <a class="btn-auth btn-facebook
large" id="login-facebook" data-loading-text="Carregando..."
href="@urlFacebook">
                            Entrar com o Facebook</b>
                        </a>
                    </div>
                </div>
            </div>
        </section>
        <div class="push"></div>
    </div>
    @footer()
}
```

MovieMe/app/views/main.scala.html

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Movie.Me - @title</title>
        <!-- Fonts -->
        <link href="http://netdna.bootstrapcdn.com/font-
awesome/4.0.3/css/font-awesome.css" rel="stylesheet">
        <link
href='http://fonts.googleapis.com/css?family=Roboto:400,100,70
0' rel='stylesheet' type='text/css'>
        <!-- Stylesheets -->
        <link
href="@routes.Assets.at("css/bootstrap.min.css")"
rel="stylesheet"/>
        <link href="@routes.Assets.at("css/style.min.css")"
rel="stylesheet"/>
        <link href="@routes.Assets.at("css/auth-buttons.css")"
rel="stylesheet">
        <script type="text/javascript"
src='@routes.RecommenderController.javascriptRoutes()'></scrip
t>
    </head>
    <body>
        @content
    </body>
</html>
```

MovieMe/app/views/movie.scala.html

```
@(movie: Movie, userName : String)
@main(movie.title_br) {
     <div class="wrapper">
          @header(userName, true, true)
          <!-- Main content -->
          <section class="container">
               <div class="col-sm-12">
                    <div class="movie">
                         <h2 class="page-
heading">@movie.title_br</h2>
                         <div class="movie-infos">
                              <div class="col-md-3">
                                   <div class="movie-image">
                                        <img
src="https://image.tmdb.org/t/p/w300@movie.url_poster">
                                   </div>
                              </div>
                              <div class="col-sm-8">
```

```
                                    <p class="movie-
runtime">@movie.runtime</p>
                                    <p class="movie-
option"><strong>Título Original:
</strong>@movie.title_original</p>
                                    <p class="movie-
option"><strong>Países: </strong>
                                        @for((movieCountry,
index) <- movie.countries.zipWithIndex) {
                                            @if(index != 0) {
                                                |
@movieCountry.country.name_br
                                            } else {

     @movieCountry.country.name_br
                                            }
                                        }
                                    </p>
                                    <p class="movie-
option"><strong>Ano: </strong>@movie.year</p>
                                    <p class="movie-
option"><strong>Gêneros: </strong>
                                        @for((movieGenre,
index) <- movie.genres.zipWithIndex) {
                                            @if(index != 0) {
                                                |
@movieGenre.genre.name_br
                                            } else {

     @movieGenre.genre.name_br
                                            }
                                        }
                                    </p>
                                    <p class="movie-
option"><strong>Diretores: </strong>
                                        @for((movieDirector,
index) <- movie.directors.zipWithIndex) {
                                            @if(index != 0) {
                                                |
@movieDirector.director.name
                                            } else {

     @movieDirector.director.name
                                            }
                                        }
                                    </p>
                                    <p class="movie-
option"><strong>Elenco: </strong>
                                        @for((movieActor,
index) <- movie.actors.zipWithIndex) {
                                            @if(index != 0) {
```

```
                                                    |
@movieActor.actor.name
                                                } else {

     @movieActor.actor.name
                                                }
                                            }
                                        </p>
                                    </div>
                                </div>
                                <div class="clearfix"></div>
                                <h2 class="page-
heading">Sinopse:</h2>
                                <p class="movie-
plot">@movie.plot_br</p>
                                <br/>
                                <a href="/movies" class="btn btn-md
btn-red">Voltar</a>
                            </div>
                        </div>
                    </section>
                    <div class="push"></div>
               </div>
               @footer()
}
```

MovieMe/app/views/recommendations.scala.html

```
@(enoughData: Boolean, hasRecommendations : Boolean,
recommendedMovies : List[RecommendedMovie], userName : String,
totalUserLikedMovies: Integer, totalUserWatchedMovies :
Integer, totalUserRatedMovies : Integer,
totalUserLikedActorsDirectors: Integer)
@main("Filmes Recomendados") {
     <div class="wrapper">
          @header(userName, enoughData, hasRecommendations)
          <!-- Main content -->
          <section class="container">
               <div class="movie">
                    <h2 class="page-heading">Filmes
Recomendados para Você...</h2>
                    @if(enoughData) {
                         @for(recommendedMovie <-
recommendedMovies) {
                              <!-- Movie -->
                              <div class="col-sm-6">
                                   <div class="movie movie-
list">
                                        <div class="row">
```

```
                                                    <div class="col-
md-5">
                                                        <div
class="movie-image">
                                                            <a
href="@routes.RecommenderController.movieDetails(recommendedMo
vie.movie.id)">

     <img alt=""
src="https://image.tmdb.org/t/p/w300@recommendedMovie.movie.ur
l_poster">
                                                            </a>
                                                        </div>
                                                    </div>
                                                    <div class="col-
md-7">
                                                        <div
class="movie-infos">
                                                            <a
href="@routes.RecommenderController.movieDetails(recommendedMo
vie.movie.id)" class="movie-title ">

     @recommendedMovie.movie.title_br
(@recommendedMovie.movie.year)
                                                            </a>
                                                            <p
class="movie-runtime">@recommendedMovie.movie.runtime</p>
                                                            <p
class="movie-option">

     @for((movieGenre, index) <-
recommendedMovie.movie.genres.zipWithIndex) {

     @if(index != 0) {

          | @movieGenre.genre.name_br

     } else {

          @movieGenre.genre.name_br

     }
                                                                }
                                                            </p>
                                                        </div>
                                                        <div
class="movie-more">
                                                            <a
id="approved-movie@recommendedMovie.movie.id"
onclick="javascript:saveUserMovieRecommendationApproved('@rout
es.RecommenderController.saveRecommendedMovieApproved(recommen
```

```
dedMovie.movie.id, 1)', @recommendedMovie.movie.id, 1)"
class="btn btn-md btn-yellow @if(recommendedMovie.approved ==
1){ disabled }">

     @if(recommendedMovie.approved == 1) {

     Recomendação Aprovada!
                                                                  }
else {

     Aprovar Recomendação
                                                                  }
                                                            </a>

     <br/><br/>
                                                            <a
id="disapproved-movie@recommendedMovie.movie.id"
onclick="javascript:saveUserMovieRecommendationApproved('@rout
es.RecommenderController.saveRecommendedMovieApproved(recommen
dedMovie.movie.id, 0)', @recommendedMovie.movie.id, 0)"
class="btn btn-md btn-red @if(recommendedMovie.approved == 0){
disabled }">

     @if(recommendedMovie.approved == 0) {

     Recomendação Reprovada!
                                                                  }
else {

     Reprovar Recomendação
                                                                  }
                                                            </a>
                                                      </div>
                                                </div>
                                          </div>
                                    </div>
                               </div>
                          }
                     } else {
                     <br/><br/>
                     <div class="alert alert-error">
                          <span class='icon-warning'></span>
                          <strong>ERRO:</strong> Dados
insuficientes para gerar recomendações. Você não
curtiu/assistiu/avaliou nenhum filme no
                          Facebook!
                     </div>
                     }
                </div>
           </section>
           @if(recommendedMovies != null) {
```

```
                @formUserOpinion(totalUserLikedMovies,
totalUserWatchedMovies, totalUserRatedMovies,
totalUserLikedActorsDirectors)
           }
     <div class="push"></div>
     </div>
    @footer()
}
```